

Effortless Data Exploration with zenvisage: An Expressive and Interactive Visual Analytics System

Tarique Siddiqui¹ Albert Kim² John Lee¹ Karrie Karahalios¹ Aditya Parameswaran¹

¹University of Illinois (UIUC) ²MIT
{tsiddiq2,lee98,kkarahal,adityagp}@illinois.edu alkim@mit.edu

ABSTRACT

Data visualization is by far the most commonly used mechanism to explore data, especially by novice data analysts and data scientists. And yet, current visual analytics tools are rather limited in their ability to guide data scientists to interesting or desired visualizations: the process of visual data exploration remains cumbersome and time-consuming. We propose zenvisage, a platform for effortlessly visualizing interesting patterns, trends, or insights from large datasets. We describe zenvisage's general purpose visual query language, ZQL ("zee-quel") for specifying the desired visual trend, pattern, or insight — ZQL draws from use-cases in a variety of domains, including biology, mechanical engineering, climate science, and commerce. We formalize the expressiveness of ZQL via a visual exploration algebra, and demonstrate that ZQL is at least as expressive as that algebra. While analysts are free to use ZQL directly, we also expose ZQL via a visual specification interface that we describe in this paper. We then describe our architecture and optimizations, preliminary experiments in supporting and optimizing for ZQL queries in our initial zenvisage prototype, and a user study to evaluate whether data scientists are able to effectively use zenvisage for real applications.

1. INTRODUCTION

The rising popularity of visual analytics tools have paved the way for the democratization of data exploration and data science. Increasingly, amateur data scientists from a variety of sectors now have the ability to explore and derive insights from data. The standard recipe for data science goes as follows: the data scientist loads the dataset into a visual analytics tool like Tableau [8], Spotfire [6] or Microsoft Excel, or even a domain-specific data exploration tool, they select specific visualizations, and then examine whether those visualizations capture desired patterns or insights. Using these tools, the data scientists can formulate and test hypotheses, and derive patterns or insights, if they are willing to generate enough visualizations and manually examine each one. The key premise of this work is that *manual examination of each visualization is simply unsustainable*, especially on large and complex datasets, where the number of visualizations grows rapidly with the dataset size (the number of records and attributes across relations). Even on moderately sized datasets, a data scientist may need to examine as many as tens of thousands of visualizations, all to test a single hypothesis, a severe impediment to data exploration.

To illustrate, we describe the challenges of several groups who have been hobbled by the ineffectiveness of current data exploration tools; our work has been in partnership with them:

Case Study 1: Advertising Data Analysis. Advertisers at ad analytics firm Turn, Inc., are often interested in examining their portfolio

of advertisements to see if their campaigns are performing as expected. For instance, an advertiser may be interested in seeing if there are any keywords that are *behaving unusually* with respect to other keywords in the Asia-Pacific region. To do this using the current visual analytics tools available at Turn, the advertiser needs to manually examine the plots of click-through rates (CTR) over time for each keyword (often hundreds to thousands of such plots).

Case Study 2: Genomic Data Analysis. Clinical researchers at NIH-funded center at the University of Illinois and Mayo Clinic are interested in studying data from clinical trials, in the context of gene and protein data. One such task involves finding pairs of genes that *visually explain the differences* in clinical trial outcomes. Current tools require the researchers to generate and manually evaluate tens of thousands of scatter plots for whether the outcomes (positive vs. negative) can be clearly distinguished in the scatter plot.

Case Study 3: Engineering Data Analysis. Battery scientists at Carnegie Mellon University perform visual exploration of datasets containing solvent properties at various scales—molecular, meso, and continuum—to design better batteries. A specific task may involve finding solvents with desired behavior: e.g., those whose solvation energy of Li^+ vs. the boiling point is an *increasing trend*. To do this using current tools, these scientists manually examine these plots for each of the thousands of solvents.

Case Study 4: Environmental Data Analysis. Climate scientists at the National Center for Supercomputing Applications at Illinois are interested in studying the nutrient and water property readings on sensors within buoys at various locations in the Great Lakes. Often, they find that a sensor is displaying unusual behavior for a specific property, and want to figure out *what is different* about this sensor relative to others, and if other properties for this sensor are *showing similar behavior*. In either case, the scientists would need to separately examine each property for each sensor (in total 100s of thousands of visualizations) to identify explanations or similarities.

Case Study 5: Server Monitoring Analysis. The server monitoring team at Facebook has noticed a spike in the per-query response time for Image Search in Russia on August 15, after which the response time flattened out. The team would like to identify if there are *other attributes that have a similar behavior* with per-query response time, which may indicate the reason for the spike and subsequent flattening. To do this, the server monitoring team generates visualizations for different metrics as a function of the date, and assess if any of them has similar behavior to the response time for Image Search. Given that the number of metrics is likely in the thousands, this takes a very long time.

Case Study 6: Mobile App Analysis. The complaints section of the Google mobile platform team have noticed that a certain mobile app has received many complaints. They would like to figure out *what is different* about this app relative to others. To do this,

they need to plot various metrics for this app to figure out why it is behaving anomalously. For instance, they may look at network traffic generated by this app over time, or at the distribution of energy consumption across different users. In all of these cases, the team would need to generate several visualizations manually and browse through all of them in the hope of finding what could be the issues with the app.

In all of these examples, the recurring theme is the *manual examination of a large number of generated visualizations for a specific visual insight*, a tedious and time-consuming process.

Our goal in this paper is to build *zenvisage*, a visual analytics system that can *automatically “fast-forward” to the desired insights*, thereby minimizing significant burden on the part of the data scientists or analysts in scenarios like the ones described above. *zenvisage* enables scientists to not just receive recommendations of interesting visualizations, but to actively request visualizations (among a large space of candidates) that convey desired trends or patterns, depict differences or explanations, and show typical or anomalous behavior, all via a few interactions, eliminating the need for tedious manual examination to perform the same task.

Given the wealth of data analytics tools available, one may ask why a new tool is needed. With these tools, selecting the right view on the data that reveals the desired insight still remains laborious and time-consuming. The onus is on the user to manually a number of visualizations until the desired one is identified. In particular, existing tools are inadequate, including: 1) *Relational databases*: Databases are powerful and efficient, but existing database query languages are too low-level to express queries like “show me visualizations of keywords where CTR over time in Asia is behaving unusually”. That said, our solution is an abstraction that sits atop traditional relational databases as a storage and computation engine. 2) *Data mining tools*: Data mining tools are challenging to use, since they require programming experience, as well as an understanding of which data mining tool applies to each task. Furthermore, the code used to express queries is often verbose and must be manually optimized (not desirable for ad-hoc querying). 3) *Visual analytics tools*: Visual analytics tools like Tableau and Spotfire have made it much easier for business analysts to analyze data; that said, the user still needs to specify exactly what they want to visualize. If the visualization does not yield the desired insight, then the user must try again, with a different visualization. One can view *zenvisage* as a substantial generalization of standard visualization specification tools like Tableau; capturing all the Tableau functionality, while providing the means to “skip ahead” to the desired insights. We describe related work in more detail in Section 8.

In this paper, we describe the specification for our query language for *zenvisage*, ZQL. We describe how ZQL is powerful enough to capture the use cases described above as well as many other use cases (Section 2). Our primary contribution in this paper is ZQL, which resulted from a synthesis of desiderata after discussions with analytics teams from a variety of domains (described above). In addition, we formalize the notion of a *visual exploration algebra*, an analog of relational algebra, describing a core set of capabilities for any language that supports visual data exploration, and demonstrate that ZQL is *complete* in that it subsumes these capabilities (Section 7). We describe query translation and execution for ZQL, and show that ZQL can leverage any relational database system as a back-end (Section 3). We then describe our initial prototype of *zenvisage* which implements a subset of ZQL, the end-user interface, as well as the underlying system architecture (Section 4). We describe our initial performance experiments (Section 5), and present a user study focused on evaluating the effectiveness and usability of *zenvisage* (Section 6).

2. QUERY LANGUAGE

zenvisage’s query language, ZQL, provides users a flexible and intuitive mechanism to specify desired insights from visualizations. The user may either directly write ZQL, or they may use the *zenvisage* front-end, which transforms all requests to ZQL internally. Our design of ZQL builds on work on visualization specification for visual analytics tools, in particular from Polaris [47], and Grammar of Graphics [51]. Indeed, *zenvisage* is intended to be a generalization of Polaris/Tableau [47], and hence must encompass Polaris functionality as well as additional ones for searching for desired trends, patterns, and insights. In addition, ZQL also draws inspiration from the Query by Example (QBE) Language [52] and uses a similar table-based interface. (However ZQL is not tied to this interface, as we describe in Section 2.1.)

Our goal for ZQL was to ensure that users would be able to effortlessly express complex requirements using a small number of ZQL lines. Furthermore, the language itself should be robust and general enough to capture the wide range of possible visual queries. As we will see later, despite the generality of the language, we have built an automatic parser and optimizer that can apply to any ZQL query and transforms it into a collection of SQL queries, along with post-processing that is run on the results of the SQL queries: this means that *zenvisage* can use as a backend any traditional relational database. To illustrate the power and the generality of the language, we now illustrate a few examples of ZQL queries, before we dive into the ZQL formalism. To make it easy to follow without much background, we use a fictitious product sales-based dataset throughout this paper in our query examples—we will reveal attributes of this dataset as we go along. Some details are omitted from the main body of the paper, and can be found in the appendix.

Query 1: Depict a collection of visualizations. Table 1 depicts a very simple ZQL query. This ZQL query retrieves the data for each product’s total sales over years bar chart visualization for products sold in the US. As the reader can probably guess, the ‘year’ and ‘sales’ in the X and Y columns dictate the x- and y- axes of the visualizations, and the location=‘US’ in the Constraints column constrains the data to items sold in the US. Then, for the Z column, we use the variable *v1* to iterate over ‘product’.*, the set of all possible product values. The *bar.(y=agg(‘sum’))* denotes that the visualization is a bar chart where the y-values are aggregated using the SUM function grouped by both the x- and z- axes. The Process column is typically used to filter, sort, or compare visualizations, but in this case, since we want the full set of visualizations (one for every product), we leave the Process column blank. This generates a separate sales vs. year plot for each product, giving a collection of resulting visualizations. This collection of visualizations is referred to using the variable *f1*, with the * indicating that these visualizations are to be output to the user. (Note that both variables *v1* and *f1* are redundant in this current query, but will come in handy for other more complex queries.) Naturally, if the number of products is large, this query could lead to a large number of visualizations being displayed, and so may not be desirable for the user to peruse. Later, we will describe mechanisms to constrain the space of visualizations that are displayed to be those that satisfy a user need. The idea that we can represent a *set* of visualizations with just one line is a powerful one, and it is part of what makes ZQL such an expressive language.

Query 2: Find the product which has the most similar sales trend as the user-drawn input trend line. Table 2 provides an example which integrates ZQL with *user-drawn* trend lines. Using *zenvis-*

Name	X	Y	Z	Constraints	Viz	Process
f1	'year'	'sales'	v1 <- 'product'.	location='US'	bar.(y=agg('sum'))	

Table 1: A ZQL query which returns the set of total sales over years bar charts for each product sold in the US.

Name	X	Y	Z	Process
-f1				
f2	'year'	'sales'	v1 <- 'product'.*	v2 <- $\text{argmin}_{v1} [k=1] D(f1, f2)$
*f3	'year'	'sales'	v2	

Table 2: A ZQL query which returns the product which has the most similar sales over year visualization as the given user-drawn trend line.

Name	X	Y	Z	Constraints	Process
f1	'year'	'sales'	v1 <- 'product'.*	location='US'	v2 <- $\text{argany}_{v1} [t > 0] T(f1)$
f2	'year'	'sales'	v1	location='UK'	v3 <- $\text{argany}_{v1} [t < 0] T(f2)$
f3	'year'	'sales'	v4 <- (v2.range & v3.range)		v5 <- $R(10, v4, f3)$
*f4	'year'	'profit'	v5		

Table 3: A ZQL query which returns the profit over years visualizations for products that have positive sales over years trends for the US but have negative sales over years trends for the UK.

age’s front-end, the user can draw a trend line¹, which ZQL can use as an input and compare against other visualizations from the database. In Table 2, we use - in the -f1 to denote that it corresponds to a visualization provided by the user. After the user input line, we see a second line which looks similar to the example in the first query; f2 iterates over the set of sales over year visualizations for each product. With the Process column, we can compare the visualizations f1 and f2 with some distance metric D for every product value in v1. argmin looks through the comparisons and selects the one product which minimizes the distance. Finally, f3 outputs the sales over year visualization for that product. Thus, with Table 2, we have managed to perform a similarity search for visualizations against a user-drawn input.

Query 3: Find and visualize profit for products that are doing well on sales in the US but badly in the UK. Finally, we look at an even more complex example in Table 3. This query captures the need to find the profit over years visualizations for products that have positive sales over years trends for the US but have negative sales over years trends for the UK, a task business users may be interested in performing. However, the way users would currently achieve this is by manually examining the sales over years charts for both the US and the UK for every product and remembering which one had the most discrepancy. With ZQL, the visual query can be expressed with three lines. The first line retrieves the set of sales over years visualizations for each product sold in the US and filters it to only include the ones in which the overall trend ($T(f1)$) is positive ($t > 0$). Likewise, the the second line retrieves the type of visualizations for products sold in the UK and filters it to only include the ones with negative ($t < 0$) overall trends ($T(f2)$). The third row combines the results of the two by taking the intersection of the sets of products ($v2.\text{range} \& v3.\text{range}$) and labels the profit over years visualizations for these products as f3. Then 10 visualizations which form the representative set of the visualizations in f3 are chosen using the function R , and returned.

Next, we go more in depth into the ZQL language and provide a formal specification. In Section 7, we provide a formalization for the expressiveness of ZQL. Additional real-world examples of how ZQL can be used can be found in Appendix C.

2.1 Formalization

We now formally describe the ZQL syntax. We assume that we are operating on a single relation or a star schema where the attributes are unique (barring key-foreign key joins). In general, ZQL could be applied to arbitrary collections of relations by letting the user precede an attribute A with the relation name R , e.g., RA . For ease of exposition, we focus on the single relation case.

¹zenvisage provides many options for user input, including directly drawing a visualization using the zenvisage front-end, providing a set of data values, or specifying a list of constraints that the visualization must satisfy.

2.1.1 Overview

As described earlier, a ZQL query is composed using a table, much like a QBE query. Unlike QBE, the columns of a ZQL query do not refer to attributes of the table being operated on; instead, they are predefined, and have fixed semantic meanings. In particular, at a high level, the columns are: (i) *Name*: providing an identifier for a set or collection of visualizations, and allowing us to indicate if a specific set of visualizations are to be output (ii) *X*, *Y*: specifying the X and Y axes of the collections of visualizations, restricted to sets of attributes (iii) *Z (optional)*: specifying the “slice” (or subset) of data that we’re varying restricted to sets of attributes along with values for those attributes; (iv) *Constraints (optional)*: specifying optional constraints applied to the data prior to any visualizations or collections of visualizations being generated (v) *Viz (optional)*: specifying the mechanism of visualization, e.g., a bar chart, scatterplot, as well as the associated transformation, or aggregation, e.g., the X axis is binned in groups of 20, while the Y axis attribute is aggregated using SUM. If this is not specified, standard rules of thumb are used to determine the appropriate visualization [28, 36]. (vi) *Process (optional)*: specifying the “optimization” operation performed on a collection of visualizations, typically intended towards identifying desired visualizations.

A ZQL query may have any number of rows, and conceptually each row represents a set of visualizations of interest. The user can then process and filtrate these rows until she is left with only the output visualizations she is interested in. The result of a ZQL query is the *data* used to generate visualizations. The zenvisage front-end then generates the visualizations for the user to peruse.

The Merits of a Tabular Interface. The reader may wonder why we chose a table as an interface for entering ZQL queries. Our choice is due to our end-users: primarily non-programmers who are used to drop-down menus and spreadsheet tools like Microsoft Excel, and feel more at home with a tabular interface. Specifically, the tabular skeleton ensures that users would not “miss out” on key columns, and can view the query on the web-client front-end as a collection of correct steps, each of which corresponds to a row. Indeed, our user study validates this point; even users with minimal programming experience can still use ZQL proficiently after a short tutorial. Additionally, this interface is much more suited for embedding into an interactive web-client. That said, a user with more experience in programming may find the interface restrictive, and may prefer issuing the query as a function call within a programming language. Nothing in our underlying ZQL backend is tied to the tabular interface: specifically, we already support the issuing of ZQL queries within our Java client library: users can easily embed ZQL queries into other computation. In the future, we plan to write wrappers for other libraries so that users can embed ZQL into computation in other settings. We are exploring the use of Thrift [46] for this purpose.

2.1.2 X and Y Columns

As mentioned, each row can be thought of as a set of visualizations, and the X and Y columns represent the x- and y- axes for those visualizations. In Table 1, the first row’s visualizations will have ‘year’ as their x-axis and ‘profit’ as their y-axis.

The only permissible entries for the X or Y column are either a single attribute from the table, or a *set* of attributes, with a variable to iterate over them. The exact semantics of variables and sets are discussed later in Section 2.1.7, but essentially, a column is allowed to take on any attribute from the set. For example, in Table 4, the y-axis is allowed to take on either ‘profit’ or ‘sales’ as its attribute. Because the y-axis value can be taken from a set of attributes, the resulting output of this query is actually the *set* of visualizations whose x-axis is the ‘time’ and the y-axis is one of ‘profit’ and ‘sales’ for the ‘stapler’. This becomes a powerful notion later in the Process column where we try to iterate over a set of visualizations and identify desired ones to return to the user or select for down-stream processing.

Name	X	Y	Constraints
*f1	‘year’	y1 <- {‘profit’, ‘sales’}	product=‘stapler’

Table 4: A query for a set of visualizations, one of which plots profit over year and one of which plots sales over time for the stapler.

2.1.3 Z Column

The Z column is used to either focus our attention on a specific slice (or subset) of the dataset or iterate over a set of slices for one or more attributes. To specify a set of slices, the Z column must specify (a) one or more attributes, just like the X and Y column, and (b) one or more attribute values for each of those attributes — which allows us to slice the data in some way. For both (a) and (b) the Z column could specify a single entry, or a variable associated with a set of entries. Table 5 gives an example of using the Z column to visualize the sales over time data specifically with regards to the ‘chair’ and ‘desk’ products, one per line. Note that the attribute name and the attribute value are separated using a period.

Table 6 depicts an example where we iterate over a set of slices (attribute values) for a given attribute. This query returns the set of sales over time visualizations for each product. Here, v1 binds to the values of the ‘product’ category. Since ZQL internally associates attribute values with their respective attributes, there is no need to specify the attribute explicitly for v1. Another way to think about this is to think of v1 <- ‘product’.* as syntactic sugar for ‘product’.v1 <- ‘product’.*. The * symbol denotes all possible values; in this case, all possible values of the ‘product’ attribute.

Name	X	Y	Z
*f1	‘year’	‘sales’	‘product’.‘chair’
*f2	‘year’	‘sales’	‘product’.‘desk’

Table 5: A ZQL query that returns the sales over year visualization for chairs and the sales over time visualization for desks.

Name	X	Y	Z
f1	‘year’	‘sales’	v1 <- ‘product’.

Table 6: A ZQL query that returns the set of sales over year visualizations for each product.

Furthermore, the Z column can be left blank if the user does not wish to slice the data in any way.

2.1.4 Constraints Column

The Constraints column is used to specify further constraints on the set of data used to generate the set of visualizations. Conceptually, we can view the Constraints column as being applied to the dataset first, following which a collection of visualizations are generated on the constrained dataset. While the Constraints column may appear to overlap in functionality with the Z column, the Constraints column does not admit iteration via variables in the way

the Z, X or Y column, admits. It is instead intended to apply a fixed boolean predicate to each tuple of the dataset prior to visualization in much the same way the WHERE clause is applied in SQL. (We discuss this issue further in Section 2.1.7.) A blank Constraints column means no constraints are applied to the data prior to visualization.

2.1.5 Viz Column

Given the data from the X, Y, Z, and Constraints columns, the Viz column determines how the data is shaped before returning it as a visualization. There are two aspects to the Viz column: first, the visualization type (e.g., bar chart, scatter plot), and the summarization type (e.g., binning or grouping, aggregating in some way). These two aspects are akin to the geometric and statistical transformation layers from the Grammar of Graphics, a visualization specification language, and the inspiration behind ggplot [50].

In ZQL, the Viz column specifies the visualization type and the summarization using a period delimiter. Functions are used to represent both the visualization type and the summarization. Consider the query in Table 7. Here, the user specifies that she wants a bar chart, with bar, and specifies the type of summarization in the accompanying tuple: x=bin(20) denotes that x-axis should be binned into bins of size 20, and y=agg(‘sum’) runs the SUM aggregation on the y-values when grouped by both the bins of the x-axis and the values in the z-axis, (if any are specified).

Often we can leave the Viz column blank. In such cases, we would apply well-known rules of thumb for what types of visualization and summarization would be appropriate given specific X and Y axes. Work on recommending appropriate visualization types dates back to the 80s [36], that both Polaris/Tableau [47], and recent work builds on [28, 29], determining the best visualization type by examining the schema and statistical properties. In many of our examples of ZQL queries, we omit the Viz column for this reason.

2.1.6 Name Column

For any row of a ZQL query, the combination of X, Y, Z, Constraints, and Viz columns together represent the *visual component* of that row. A visual component formally represents a set of visualizations. The Name column allows us to provide a name to this visual component by binding a variable to it. These variables can be used in the Process column to subselect the desired visualizations from the set of visualizations, as we will see subsequently.

In the ZQL query given by Table 8, we see that the names for the visual components of the rows are named f1, f2, and f3 in order of the rows. For the first row, the visual component is a single visualization, since there are no sets, and f1 binds to the single visualization. For the second row, we see that the visual component is over the set of visualizations with varying Z column values, and f2 binds to the variable which iterates over this set. Note that f1 and f3 in Table 8 are prefaced by a * symbol. This symbol indicates that the visual component of the row is designated to be part of the output. As can be seen in the example, multiple rows can be part of the output, and in fact, a row could correspond to multiple visualizations. Visualizations corresponding to all the rows marked with * are processed and displayed by the zenvisage frontend.

2.1.7 Sets and Variables

As we described previously, sets in ZQL must always be accompanied by a variable which iterates over that set, to ensure that ZQL traverses over sets of visualizations in a consistent order when making comparisons. Consider Table 10, which shows a query that iterates over the set of products, and for each, compares the sales over

Name	X	Y	Viz
*f1	'weight'	'sales'	bar.(x=bin(20), y=agg('sum'))

Table 7: A ZQL query which returns the bar chart of overall sales for different weight classes.

Name	X	Y	Z	Process
*f1	'year'	'sales'	'product'.'stapler'	v2 <- argmin _{v1} [k = 10]D(f1, f2)
f2	'year'	'sales'	v1 <- 'product'.(* - 'stapler')	
*f3	'year'	'sales'	v2	

Table 8: A ZQL query retrieving the sales over year visualization for the top 10 products whose sales over year visualization looks the most similar to that of the stapler.

Name	X	Y	Z	Process
-f1	x1 <- {'time', 'location'} x2	y1 <- {'sales', 'profit'} y2	'product'.'stapler' 'product'.'stapler'	x2, y2 <- argmin _{x1,y1} [k = 10]D(f1, f2)
f2				
*f3				

Table 9: A ZQL query retrieving two different visualisations (among different combinations of x and y) of stapler which are most similar to each other

Name	X	Y	Z	Process
f1	'year'	'sales'	v1 <- 'product'.*	v2 <- argmax _{v1} [k = 10]D(f1, f2)
f2	'year'	'profit'	v1	
*f3	'year'	'sales'	v2	
*f4	'year'	'profit'	v2	

Table 10: A ZQL query which returns the set sales over years and the profit over years visualizations for the top 10 products for which these two visualizations are the most dissimilar.

Name	X	Y	Z	Constraints	Process
f1	'year'	'sales'	v1 <- 'product'.*	product IN (v2.range)	v2 <- argmax _{v1} [k = 10]T(f1)
*f2	'year'	'profit'			

Table 11: A ZQL query which plots the profit over years for the top 10 products with the highest sloping trend lines for sales over the years.

Name	X	Y	Z	Process
f1	x1 <- C	y1 <- M	'product'.'chair'	x2, y2 <- argmax _{x1,y1} [k = 10]D(f1, f2)
f2	x1	y1	'product'.'desk'	
*f3	x2	y2	'product'.'chair'	
*f4	x2	y2	'product'.'desk'	

Table 12: A ZQL query which finds the x- and y- axes which differentiate the chair and the desk most.

years with the profits over years. Without a variable enforcing a consistent iteration order over the set of products, it is possible that the set of visualizations could be traversed in unintended ways. For example, the sales vs. year plot for chairs from the first set could be compared with the profit vs. year plot for desks in the second set. By reusing v1 in both the first and second rows, the user can force the zenvisage back-end to step through the sets in sync.

Operations on Sets. Constant sets in ZQL must use {} to denote that the enclosed elements are part of a set. As a special case, the user may also use * to represent the set of all possible values. The union of sets can be taken using the sign |, set difference can be taken using the \ sign, and the intersection can be taken with &.

Ordering. All sets in zenvisage are ordered, but when defined using the {}, the ordering is defined arbitrarily. Only after a set has passed through a Process column's ordering mechanism can the user depend on the ordering of the set. Ordering mechanisms are discussed further in Section 2.1.8. However, even for arbitrarily ordered sets, if a variable iterator is defined for that set and reused, ZQL guarantees at least a *consistent* ordering of traversal, allowing the sets in Table 10 to be traversed in the intended order.

Axis Variables. In ZQL, there are two types of variables: *axis variables* and *name variables*. Axis variables are the common variables used in any of the columns except the Name column. The declaration has the form: $\langle \text{variable name} \rangle \leftarrow \langle \text{set} \rangle$. The variable then can be used as an iterator in the Process column or reused in a different table cell to denote that the set be traversed in the same way for that cell. It is possible to declare multiple axis variables at once, as we have seen from the Z and Process columns: (e.g., $z.v \leftarrow *.*$).

Sometimes, it is necessary to retrieve the set that an axis variable iterates over. The .range notation allows the user to expand variables to their corresponding sets and apply arbitrary zenvisage set operations on them. For example, $v4 \leftarrow (v2.range \mid v3.range)$ binds v4 to the union of the sets iterated by v2 and v3.

Axis variables can be used freely in any column except the Constraints column. In the Constraints column, only the expanded set

form of a variable may be used. Table 11 demonstrates how to use an axis variable in the Constraints column. In this example, the user is trying to plot the overall profits over years for the top 10 products which have had the most growth in sales over the years. The user finds these top 10 products in the first row and declares the variable v2 to iterate over that set. Afterwards, she uses the constraint $\text{product} \leftarrow (v2.range)$ to get the overall profits across all 10 of these products in the second row.

Name Variables. Name variables (e.g., f1, f2) are declared only in the Name column and used only in the Process column. Named variables are iterators over the set of visualizations represented by the visual component. If the visual component represents only a single visualization, the name variable is set to that specific visualization. Name variables are *bound* to the axis variables present in their row. In Table 11, f1 is bound v1, so if v1 progresses to the next product in the set f1 also progresses to the next visualization in the set. This is what allows the Process column to iterate over the axis variable (v1), and still compare using the name variable (f1). If multiple axis variables are present in the visual component, the name variable iterates over set of visualizations produced by the Cartesian product of the axis variables. If the axis variables have been declared independently of each other in the visual component, the ordering of the Cartesian product follows the ordered bag semantics described in Section 7.2 in order of the columns laid out in ZQL. However, the user may also superscript variables to control the order in which Cartesian product is done as well. However, if the variables were declared together in the Process column as is the case with x2 and y2 in Table 12, the ordering is the same as the set in the declaration.

2.1.8 Process Column

Once the visual component for a row has been named, the user may use the Process column to sort, filter, and compare the visual component with previously defined visual components to isolate the set of visualizations she is interested in. While the user is free

Name	X	Y	Z	Process
f1	'year'	'sales'	v1 <- 'product'.*	v2 <- R(10, v1, f1)
f2	'year'	'sales'	v2	v3 <- argmax _{v1} [k = 10] min _{v2} D(f1, f2)
*f3	'year'	'sales'	v3	

Table 13: A ZQL query which returns 10 sales over years visualizations for products which are outliers compared to the rest.

to define her own functions for the Process column, we have come up with a core set of primitives based on our case studies which we believe can handle the vast majority of the common operations. Each non-empty Process column entry is defined to be a *task*. Implicitly, a task may be impacted by one or more rows of a ZQL query (which would be input to the process optimization), and may also impact one or more rows (which will use the outputs of the process optimization), as we will see below.

Functional Primitives. First, we introduce three simple functions that can be applied to visualizations: T , D , and R . `zenvisage` will use default settings for each of these functions, but the user is free to specify their own variants for each of these functions that are more suited to their application.

- $T(f)$ measures the overall trend of visualization f . It is positive if the overall trend indicates “growth” and negative if the overall trend goes “down”. There are obviously many ways that such a function can be implemented, but one example implementation might be to measure the slope of a linear fit to the given input visualization f .

- $D(f, f')$ measures the distance between the two visualizations f and f' . For example, this might mean calculating the Earth Mover’s Distance or the Kullback-Leibler Divergence between the induced probability distributions.

- $R(k, v, f)$ computes the set of k -representative visualizations given an axis variable, v , and an iterator over the set of visualizations, f . Different users might have different notions of what representative means, but one example would be to run k -means clustering on the given set of visualizations and return the k centroids. In addition to taking in a single axis variable, v , R may also take in a tuple of axis variables to iterate over. The return value of R is the set of axis variable values which produced the representative visualizations.

Processing. Given these functional primitives, ZQL also provides some default sorting and filtering mechanisms: `argmin`, `argmax`, and `argany`. Although `argmin` and `argmax` are usually used to find the best value for which the objective function is optimized, ZQL typically returns the top- k values, sorted in order. `argany` is used to return any k values. In addition to the top- k , the user might also like to specify that she wants every value for which a certain threshold is met, and ZQL is able to support this as well. Specifically, the expression `v2 <- argmaxv1[k = 10]D(f1, f2)` returns the top 10 $v1$ values for which $D(f1, f2)$ are maximized, sorts those in decreasing order of the distance, and declares the variable $v2$ to iterate over that set. This could be useful, for instance, to find the 10 visualizations with the most variation on some attributes. The expression `v2 <- argminv1[t < 0]D(f1, f2)` returns the $v1$ values for which the objective function $D(f1, f2)$ is below the threshold 0, sorts the values in increasing order of the objective function, and declares $v2$ to iterate over that set. If a filtering option (k, t) is not specified, the mechanism simply sorts the values, so `v2 <- argminv1T(f1)` would bind $v2$ to iterate over the values of $v1$ sorted in increasing order of $T(f1)$. `v2 <- arganyv1[t > 0]T(f1)` would set $v2$ to iterate over the values of $v1$ for which the $T(f1)$ is greater than 0.

Note that the mechanisms may also take in multiple axis variables, as we saw from Table 12. The mechanism iterates over the Cartesian product of its input variables. If a mechanism is given k axis variables to iterate over, the resulting set of values must also be bound to k declared variables. In the case of Table 12, since there were two variables to iterate over, $x1$ and $y1$, there are two output variables, $x2$ and $y2$. The order of the variables is important as the values of the i th input variable are set to the i th output variable.

User Exploration Tasks. By building on these mechanisms, the user can perform most common tasks. This includes the *similarity/dissimilarity search* performed in Table 10, to find products that are dissimilar; the *comparative search* performed in Table 12, to identify attributes on which two slices of data are dissimilar, or even the *outlier search* query shown in Table 13, to identify the products that are outliers on the sales over year visualizations. Note that in Table 13, we use two levels of iteration.

3. QUERY EXECUTION

In `zenvisage`, ZQL queries are automatically parsed and executed by the backend. The ZQL compiler translates ZQL queries into a collection of SQL queries that are issued to a relational database and performs post-processing computation on the results of the SQL queries. By translating to SQL, we do not tie ourselves down to any specific relational database and can seamlessly leverage benefits from improvements to databases.

3.1 Translation to Database Queries

Each row of a ZQL query can be broken up into two parts: the visual component and the the Process column. The visual components determine the set of data `zenvisage` needs to retrieve from the database, and the Process column translates into the post-processing computation to be done on the returned data. `zenvisage`’s naive ZQL compiler performs the following operations for each row of a ZQL query. For each row, the ZQL compiler issues a SQL query corresponding to each visualization in the set specified in the visual component. Notice that this approach is akin to what a visual analyst manually generating each visualization of interest and perusing it would do; here, we are ignoring the human perception cost. More formally, for each row (i.e., each name variable), the ZQL compiler loops over all combinations of values for each of the n axis variables in the visual component corresponding to that row, and for each combination, issues a SQL query, the results of which are stored in the corresponding location in an n dimensional array. (Essentially, this corresponds to a nested for loop n levels deep.) Each generated SQL query has the form: `SELECT X, Y FROM R WHERE Z=V AND (CONSTRAINTS) ORDER BY X`. If the summarization is specified, additional clauses such as `GROUP BY`s and aggregations may be added. The results of each visualization are stored in an n -dimensional array at the current index. If n is 0, the name variable points directly to the data. Going forward, we will omit the `FROM` clause: this is implied and always fixed.

Once the name variable array has been filled in with results from the SQL queries, the ZQL compiler generates the post-processing code from the task in the Process column. At a high level, ZQL loops through all the visualizations, and applies the objective function on each one. In particular, for each input axis variable in the mechanism, a for loop is created and nested. The input axis variables are then used to step through the arrays specified by the name variable, and the objective function is called at each iteration. Instances of name variables are updated with the correct index based on the axis variables it depends on. The functions themselves are considered black boxes to the ZQL compiler and are unaltered. We provide the pseudocode of the compiled version of the query expressed in Table 14 in Listing 1 in the appendix.

3.2 External Optimizations

Name	X	Y	Z	Constraints	Viz	Process
f1	'year'	'sales'	$v1 <- P$	location='US'	bar.(y=agg('sum'))	$v2 <- \text{argmax}_{y,v1} [t > 0] T(f1)$
f2	'year'	'sales'	$v1$	location='UK'	bar.(y=agg('sum'))	$v3 <- \text{argmax}_{y,v1} [t < 0] T(f2)$
*f3	'year'	'profit'	$v4 <- (v2.\text{range} \mid v3.\text{range})$		bar.(y=agg('sum'))	

Table 14: A ZQL query which returns the profit over time visualizations for products in user-specified set P that have positive sales over time trends for the US but have negative sales over time trends for the UK.

Name	X	Y	Z	Constraints	Viz	Process
f1	'location'	'sales'	$v1 <- P$	year='2010'	bar.(y=agg('sum'))	$v2 <- \text{argmax}_{y,v1} [k = 10] D(f1, f2)$
f2	'location'	'sales'	$v1$	year='2015'	bar.(y=agg('sum'))	
*f3	'location'	'profit'	$v2$	year='2010'	bar.(y=agg('sum'))	
*f4	'location'	'profit'	$v2$	year='2015'	bar.(y=agg('sum'))	

Table 15: A ZQL query which returns the profit over location visualizations for products in user-specified set P that have the most different sales over location trends between 2010 and 2015.

Processing each visualization as an independent SQL query can be both wasteful and lead to unacceptable latencies. We now describe how to rewrite and batch the SQL queries via three levels of optimizations to reduce latency. These optimizations are reminiscent of multi-query optimization (MQO) [44]; however, MQO techniques require significant changes to the underlying database engine [20, 26, 30, 21], whereas our syntactic rewriting techniques operate completely outside the database and are tailored to the ZQL setting.

Intra-Line Optimization. The first level of optimization batches the SQL queries for a row into one query. For example, we see that for the first row in Table 14, that a separate query is being made for every product in Listing 1 in the appendix. Instead, we can retrieve the data for all products in one SQL query:

```
SELECT year, SUM(sales), product
WHERE product IN P and location='US'
GROUP BY product, year
ORDER BY product, year
```

If the axis variable is in either the X or Y columns, we retrieve the data for the entire set of attributes the axis variable iterates over. More concretely, if we have axis variable $y1 <- \{\text{'sales'}, \text{'profit'}\}$, our SQL query would look like:

```
SELECT year, SUM(sales), SUM(profit), product
WHERE product IN P and location='US'
GROUP BY product, year
ORDER BY product, year
```

This optimization cuts down the number of queries by the sizes of sets the axis variables range over, and therefore will lead to substantial performance benefits, as we will see in the experiments. Note that a side effect of batching queries is that the compiled code must now have an extra phase to extract the data for different visualizations from the combined results. However, since the ZQL compiler includes an ORDER BY clause, the overhead of this phase is minimal.

Intra-Task Optimization. In addition to combining SQL queries within a row, SQL queries may also be batched across ZQL rows. However, in general, it may not be possible to compose queries across rows into a single SQL query, since different rows may access completely different attributes. Instead, our optimization is to batch multiple SQL queries from different rows into a single *request* to the database, effectively pipelining the data retrieval. However, tasks in ZQL frequently filter and limit the space of visualizations being looked at. Moreover, visualizations from subsequent rows may depend on the output values of previous tasks, so it is not possible to batch queries across tasks. Therefore, we batch into a single request all SQL queries for task-less rows leading up to a row with a task. In Table 15, this optimization would batch rows 1 and 2 together and rows 3 and 4 together.

Inter-Task Optimization. It is not generally not possible to batch SQL queries across tasks, since subsequent tasks may depend on the outputs of previous tasks. However, for more sophisticated ZQL queries, it may be possible to identify batching opportunities

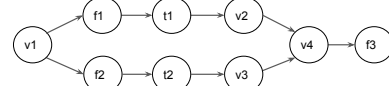


Figure 1: The query tree for the ZQL query in Table 14.

across tasks. For instance, a visual component defined after a task may be independent of the task. Formally, if visual component V is defined in a row later than the row task T is defined in, and no column of V depends on the output of T , then V is independent of T . The advantage of independence is that now we can batch V into an earlier request because we do not need to wait for the results of T . Thus, in Table 14, we can batch the SQL queries for the first and second rows into a single request since the visual component in the second row is independent of the task in the first row.

To determine all cases for which this independence can be taken advantage of, the ZQL compiler builds a query tree for the ZQL queries it parses. All axis variables, name variables, and tasks of a ZQL query are nodes in its query tree. Name variables become the parents of the axis variables in its visual component. Tasks become the parents of the visualizations it operates over. Axis variables become the parents over the nodes which are used in its declaration, which may be either tasks nodes or other axis variable nodes. The query tree for Table 14 is given in Figure 1. Here, the children point to their parents, and the tasks in rows 1 and 2 are labeled $t1$ and $t2$ respectively. As we can clearly see from the tree, the visual component for $f2$ is independent of $t1$.

The zenvisage execution engine uses the query tree to determine which SQL queries to batch in which step. At a high level, all SQL queries for name variable nodes whose children have all been satisfied or completed can be batched into a single request. More specifically, the execution engine starts out by coloring all leaf nodes in the query tree. Then, the engine repeats the following until the entire query tree has been colored: i) Batch the SQL queries for name variable nodes, who have all their children colored, into a single request and submit to the database. ii) Once a response is received for the request, color all name variables nodes whose SQL queries were just submitted. iii) Until no longer possible, color all axis variable and task nodes whose children are all colored. If a task is about to be colored, run the task first. This execution plan ensures the maximal amount of batching is done while still respecting dependencies. Parts of the execution plan may also be parallelized (such as simultaneously running two independent, computation-expensive tasks like representative sets), to improve overall throughput.

While this optimization may seem like overkill for the short ZQL examples presented in this paper, nothing prevents the user from combining many non-related visual queries into a single ZQL query with many output rows. Particularly for the case of exploratory visual analysis, the user may want to submit one ZQL query which contains many different tasks to explore interesting trends, representative sets, and outliers in an unknown dataset.

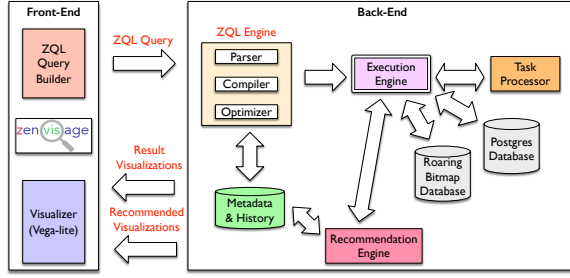


Figure 2: System architecture

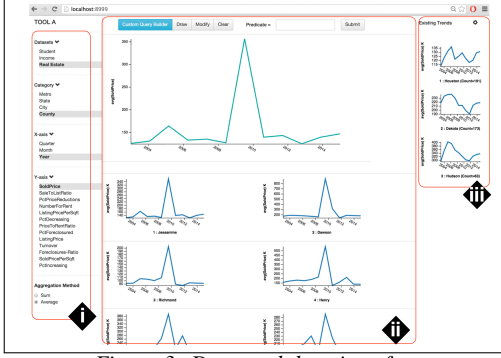


Figure 3: Drag and drop interface

4. zenvisage SYSTEM ARCHITECTURE

We now describe the architecture of zenvisage, shown in Figure 2.

4.1 Front-End

The zenvisage front-end is designed as a lightweight web-based client application, accessible via a browser on desktop, tablet, or mobile device. It performs two major functions: First, it provides the analyst an intuitive graphical interface to compose ZQL queries for exploring trends and insights in data. Second, it takes the results of these queries from the back-end and encodes them into the most effective visualizations, taking into consideration data properties and perceptual principles.

The zenvisage system is intended for both novice and expert users, and thus the interface is designed for both usage styles. Figures 3 and 4 show screenshots of our current front-end implementation (additional screenshots, displaying the full range of functionality, via Figures 12 and 13, can be found in the appendix). The interface is divided into three major components: the building blocks panel on the left (Figure 3i), the main panel in the center (Figure 3ii), and the recommendation panel on the right (Figure 3iii)—which automatically provides interesting visualizations for the axes that the user is currently viewing. The main panel is used for query building and output visualization. The query builder component consists of two parts: the drawing box (not visible in Figure 4 but available similar to Figure 3ii) and the ZQL custom

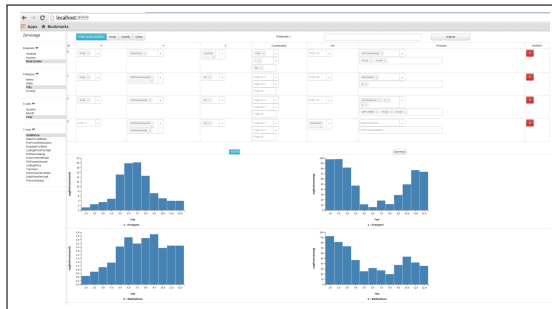


Figure 4: Custom query builder interface

query builder (Figure 4iv). Users can drag and drop any attribute from from the building blocks panel on to the x-, y- or z- axis placeholders on the drawing box. They can either directly draw the trend line, box chart, or scatterplot they are looking for from scratch or drag and drop trends from other already rendered visualizations to modify it. In addition to the drawing, the user must also specify the intended insights and trends that she is looking for. Some common data exploration queries for identifying insights and trends, such as similarity search or representative search (see Section 2.1.8), are built in to the system and exposed on the building blocks panel. (So for these data exploration queries, the user does not even need to compose ZQL queries; simply clicking the right button will do.) Novice and expert users alike can use the drawing box to easily explore the data using common exploration tasks. The ZQL front-end internally translates the selections in the drawing into a ZQL query and submits it to the back-end for execution. We call this the *drag and drop interface* of zenvisage.

Custom Query Builder. If the user would like the full expressive power of ZQL, the user may choose to use the front-end’s custom query builder, which allows users to directly specify their query in the ZQL query format, depicted in Figure 4. The builder contains all the columns necessary for writing a ZQL query. Users write their query in a row-wise manner. For each row, they can either drag and drop the attributes from the building blocks panel on to a cell in the ZQL table. If a row requires a user-drawn input, the user may select the row and use the drawing box to draw the trend she is looking for. Iterators and outputs of previous rows may also be reused by dragging and dropping them to the current cell. We call this the *custom query builder interface* of zenvisage.

Example Queries. Here, we briefly describe the queries for which the screenshots were constructed. (i) A real estate agent notices an interesting peak between 2008 and 2012 in the county of Jessamine, and now wants to discover other counties with a similar pattern during the same time frame (Figure 3). (ii) Among all the cities in NY where the selling price of houses have increased from 2004 to 2015, the agent is interested in learning about those cities where foreclosures and the rate of increase of prices followed opposite trends. In other words, she wanted to know the cities where low increase in price rates were the main reason of high foreclosures, and vice versa (Figure 4). She also wanted to see how both of these trends varied over the years. (iii) The agent is interested in finding cities in the state of NY where the recent selling price trend of houses is very different from the overall selling price trend for NY state (Figure 12 in the appendix). (iv) The agent wants to learn about states where the turnover rate (% of houses that were sold) followed the opposite pattern to housing sale prices. Generally, as the price of houses increases, turnover rate increases as well, i.e., more people tend to sell their houses. However, the agent wants to know about the states which did not follow this pattern (Figure 13 in the appendix).

Recommendation Panel. In addition to returning results for user-submitted queries, zenvisage runs a host of parallel queries to find the most interesting trends for that subset of data the user is currently viewing and presents them in the right panel. These visualizations can be considered interesting visualization recommendations, as opposed to direct answers to ZQL queries; the mechanism for this is discussed in the next section.

Result Visualizer. The zenvisage front-end’s visualizer makes use of Vega-lite [28] grammar and the Vega visualization specification for mapping the query results to effective visualizations. The zenvisage front-end determines the default visual encodings, such as as size and position, and uses the visualizer to map the output data according to the Vega-lite grammar.

4.2 Back-End

The zenvisage front-end interacts with the back-end via a REST protocol. The back-end is implemented in Java and uses `node.js` for the web server. The back-end is comprised of the following components: the ZQL Engine, Execution Engine, and Recommendation Engine. **ZQL Engine.** ZQL Engine is responsible for parsing, compiling, and optimizing given ZQL queries. See Section 3 for extensive discussion on the inner workings of this component.

Execution Engine. The Execution Engine takes the compiled output of the ZQL Engine and issues SQL queries to the database back-end while handing off post-processing on the resulting data to the task processor. We currently support two database back-ends: PostgreSQL and our own Roaring Bitmap Database.

PostgreSQL. Our current implementation uses PostgreSQL as a database back-end. However, since our ZQL Engine outputs standard SQL queries, any other relational database could be used.

Roaring Bitmap Database. Since the queries in our system are mostly ad-hoc and unpredictable, we cannot pre-compute and store query results in advance. Nor can we apply conventional indexes like B-trees as they result in high memory consumption and computation overhead for interactive use. To address these problems, we have developed a new storage model which uses Roaring Bitmaps [14] as its principal data storage format. By exploiting bit-level parallelism, bitmaps can significantly accelerate queries involving arbitrary and complex selection predicates and aggregation. Further, Roaring Bitmap is an improvement over conventional bitmaps and has 4-10X faster and higher compression rates. In our storage model, we follow a column oriented storage model. Columns which are not indexed are stored as an array on disk, and columns which are indexed have a Roaring Bitmap in memory for every distinct value of that column. As a default policy, we create Roaring Bitmaps for all categorical columns and leave measure columns un-indexed. Because of the fast bit-level parallelism and filtering capabilities, the Roaring Bitmap Database led to speedups of 30% – 50% for queries with 10% selectivity.

Task Processor. As discussed in Section 3, the Task Processor performs the post-processing computation on the results of the SQL queries. For final output data, the processor also serializes the data into JSON before handing it back to the front-end.

Recommendation Service. In addition to query results, the zenvisage back-end also maintains a set of interesting recommended visualizations to help user further understand the trends in the data. Currently we define interesting trends as those which reflect the most diversity in the data related to the user query. We identify diverse trends using on a set of heuristics. For instance, if the user is looking for a specific product that matches her drawn profit over year trends, the Recommendation Service also returns profit over trends for other products, which are diverse. In order to find the diverse trends, we run the k -means clustering algorithm to find a set of k diverse clusters in the data. By default, zenvisage sets k as 5, but the user has the options to change this value. While our current implementation is primitive, we plan to explore alternative schemes for recommendations in future work.

5. EXPERIMENTAL STUDY

We evaluate several properties of our zenvisage prototype. First, we evaluate the impact of the optimizations described in Section 3. Second, we evaluate the performance of the task processor for three common tasks often used in ZQL queries. We have also performed experiments where we compare our two backend engines, whose results can be found in Appendix D.

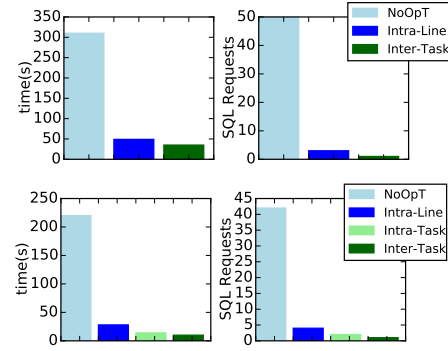


Figure 5: Runtimes for the query in Table 14 (top) and 15 (bottom)

For our experiments, we use the following datasets: (i) A synthetic sales dataset with 10M rows and with the following 8 attributes: product, size, weight, city, country, category, month, year, profit, and revenue. (ii) A real census-income dataset [2] consisting of 300,000 rows and 40 attributes. (iii) A real airline dataset [1] with 15 million rows and 29 attributes. We performed our experiments on a machine with 20 cores of Intel(R) Xeon(R) CPU E5-2680 v2 @ 2.80GHz. Our entire code-base was developed in Java.

5.1 Effect of Query Optimizations

In these experiments, we measure the improvement in performance by applying the three optimizations discussed in Section 3. We run the two ZQL queries mentioned in Section 3 (Tables 14 and 15) on the synthetic dataset and two additional queries (Tables 16 and 17) on the real airline dataset. The two additional queries respectively express a user need to find either specific patterns (find airports with increasing delay), or anomalous patterns (find airports where the discrepancies between two visualizations is maximized). Figures 5(top) and 5(bottom) show the total runtime including the SQL execution time and the computation time and the number of SQL requests made for Tables 14 and 15 on the synthetic dataset, and Figures 6(top) and 6(bottom) show the total runtime and number of SQL requests for Tables 16 and 17 on the real dataset. NoOpT is the runtime using the naive ZQL compiler. Intra-Task times are not shown for Figures 14 and 16 because they provide no opportunities for the optimization.

Results. We found that the SQL execution time dominated the overall runtime. Even when on increasing the number of products in P and airports in OA and DA (see query), the post-processing time ($< 100\text{ms}$) was negligible to the query execution time ($> 1\text{s}$).

Therefore, we see that our optimizations which reduce the number of SQL requests made to the database provide *significant speedups in overall runtime*. Specifically, the intra-line optimization has the most effect because it batches the most number of SQL queries into one. In the case of Table 14, there were 20 different products in P, and the intra-line optimization combined these 20 separate SQL queries into a single one. The other optimizations also provide benefits, but because the number of rows in this examples are so few, the improvements are marginal. We expect these optimizations to be more noticeable in large ZQL queries with many unrelated tasks.

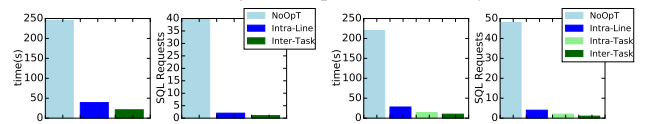


Figure 6: Runtimes for queries in Table 16 (left) and 17 (right).

5.2 Performance of the Task Processors

In this experiment, we want to evaluate how quickly the task processors return visualizations relative to the total time taken. We

Name	X	Y	Z	Constraints	Process
f1	'year'	'DepDelay'	$v1 <- OA$		$v2 <- \text{argany}_{v1}[t > 0]T(f1)$
f2	'year'	'WeatherDelay'	$v1$		$v3 \text{ IN } \text{argany}_{v1}[t > 0]T(f1)$
*f3	'year'	$y3 <- \text{'DepDelay', 'WeatherDelay'}$	$v4 <- (v2.\text{range} \mid v3.\text{range})$		

Table 16: A ZQL query which returns the departure delay over year and weather delay over year visualizations for airports in OA ({JFK,SFO...}) where the average departure or weather delay has been increasing over the years.

Name	X	Y	Z	Constraints	Process
f1	'Day'	'ArrDelay'	$v1 <- DA$	Month="06"	$v2$
f2	'Day'	'ArrDelay'	$v1$	Month="12"	$v2 <- \text{argmax}_{k=10} D(f1, f2)$
*f3	'Months'	$y1 <- \text{'ArrDelay', 'WeatherDelay'}$	$v2$		

Table 17: A ZQL query which returns the arrival delay over year and weather delay over year visualizations for airports DA ({JFK,SFO...}) where the average arrival delay between 'June' and 'Dec' differs the most.

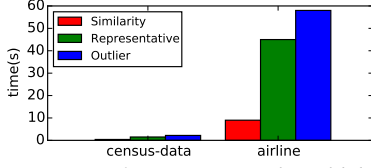


Figure 7: Performance on real world data

identify limitations or bottlenecks in our current zenvisage prototype, if any. We evaluate the total elapsed time, the total computation time, and the SQL query execution time for three types of ZQL queries: (i) *Similarity Search Query*. Here, we evaluate a query similar to Table 8, where we want to find a Z attribute value that for which a given visualization is most similar to one drawn by the user or selected up front, with ℓ_2 as a distance metric D to evaluate similarity. (ii) *Representative Search Query*. Here, we evaluate a query similar to the line corresponding to f1 in Table 13, where we want to find k Z attribute values for which the corresponding X vs. Y visualization is representative of the rest. For the function R , we use k -means clustering using the same D , and then return the cluster centroids. (iii) *Outlier Search Query*. Here, we evaluate a query similar to the entirety of Table 13, where we want to find k Z attribute values for which the corresponding X vs. Y visualization is anomalous relative to the rest. We first apply the representative search task, and then return the k visualizations for which the minimum distance D to the representative trends is maximized.

To evaluate the task processors in evaluating these ZQL queries, we measure the performance as a function of the number of groups, where groups is simply the product of the number of distinct values of the X attribute and the number of distinct values of the Z attribute—notice that these are precisely the number of distinct groups we would have in the GROUP BY query we would issue to the back-end database for each of these queries using summarization. Recall that each distinct value of the Z attribute leads to an additional visualization under consideration for that task. In case of the synthetic dataset, the number of groups are varied by changing the number of unique values of the given Z attribute, while the size of the dataset is kept fixed at 10M.

Results. Figure 8 depicts the results on all three metrics that we evaluate (in milliseconds). As seen in Figure 8(a), the overall processing time increases as we increase the number of groups. This is not surprising since there is an increase in both computation as well as query execution time. The query execution time for all the methods is similar as they all need to fetch the same amount of data from the table, but the slight increase as the number of groups increases results from the increase in the number of groups in GROUP BY of the associated SQL query. The computation time varies in proportion to the number of pairs of visualizations a given task processor compares, and therefore increases as we increase the number of groups (and therefore visualizations, as seen in Figure 8(b)). Furthermore, similarity search has the least cost of computation while outlier search has the maximum as it internally uses both representative and similarity methods. For a small number of groups, the query execution dominates the overall time. As the number of

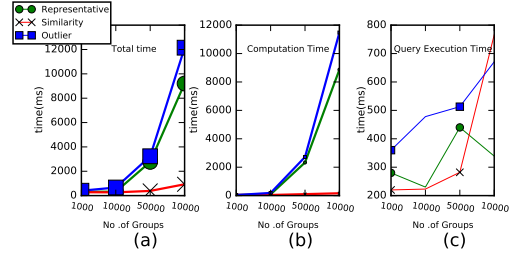


Figure 8: Performance on varying no. of groups

groups increases, the computation cost increases much faster than the query execution time especially in the case of representative and outlier search. Figure 7 shows the overall performance of the three tasks on real datasets, wherein the same relative behavior holds between the three task processors and therefore queries. In case of real datasets, since the number of groups is small, the overall time is dominated by the query execution time(>95%).

6. USER STUDY

We conducted a mixed methods study [17] in order to assess the efficacy of zenvisage. We wanted to: a) examine and compare the usability of the drag and drop interface and the custom query builder interface of zenvisage; b) compare the performance and usefulness of zenvisage with existing tools; c) identify if, when, and how people would use zenvisage in their workflow; and d) receive general feedback for future improvements.

6.1 User Study Methodology

Participants. We recruited 12 graduate students as participants with varying degrees of expertise in data analytics. Table 18 depicts the participants' experience with different categories of tools.

Tools	Count
Excel, Google spreadsheet, Google Charts	8
Tableau	4
SQL, Databases	6
Matlab,R,Python,Java	8
Data mining tools such as weka, JNP	2
Other tools like D3	2

Table 18: Participants' prior experience with data analytic tools

Dataset. We used a housing dataset from Zillow.com [9] consisting of housing sales data for different cities, counties, and states from 2004–15, with over 245K rows, and 15 attributes. We selected this dataset for two reasons: First, housing data is often explored by data analysts using existing visualization tools [18]. Second, looking for housing is commonplace for graduate students while in school and upon graduation; the participants could relate to the dataset and understand the usefulness of the tasks.

Comparison Points. There are no tools that offer the same functionalities as zenvisage. Visual analytics tools do not offer the ability to search for specific patterns, or issue complex visual exploration queries; data mining toolkits do not offer the ability to search for visual patterns and are instead tailored for general machine

learning and prediction. Since visual analytics tools are closer in spirit and functionality to zenvisage, we decided to implement a visual analytics tool as our baseline. Thus, our baseline tool replicated the basic query specification and output visualization capabilities of existing tools such as Tableau. We augmented the baseline tool with the ability to specify an arbitrary number of filters, allowing users to use filters to drill-down on specific visualizations. This baseline visualization tool was implemented with a styling scheme similar to zenvisage to control for external factors. As depicted in Figure 9, the baseline allowed users to visualize data by allowing them to specify the x-axis, y-axis, category, and filters. The baseline tool would populate all the visualizations, which fit the user specifications, using an alpha-numeric sort order. In addition to task-based comparisons with this baseline, we also explicitly asked participants to compare zenvisage with existing data mining and visual analytics tools that they use in their workflow.

Study Protocol. The user study was conducted using a within-subjects study design [12]. The study consisted of three phases. In the first phase, participants described their previous experience with data analytics and visualization tools. In the second phase, participants performed visual analytics tasks using each of the tools. zenvisage was introduced as Tool A and the baseline tool was introduced as Tool B. This phase began with a 15-minute tutorial that included an overview of the interface and two practice questions. The order of the baseline tool and the zenvisage interface was randomized to reduce order effects. To compare the effectiveness of the drag and drop interface and the custom query builder interface of zenvisage, participants performed two separate sets of tasks on zenvisage, one for each component. Finally, in the third phase, participants completed a survey that measured their satisfaction levels and preferences, followed by open-ended interview questions assessing the strengths and weaknesses of the tools, and whether they would use the two tools in their workflow. The average study session lasted for 75 minutes on average. Participants were paid ten dollars per hour for their participation.

Tasks. We prepared three sets of tasks for the three interfaces (baseline, drag and drop, and custom query builder interfaces). The tasks differed only in the selection attributes so that the performance of the interfaces could be measured objectively (Section 4.1 lists a few example queries that were used in the study). We carefully selected tasks which can surface during a search for a house. To demonstrate the efficacy of zenvisage, the tasks focused on identifying patterns, trends or anomalies in housing data; these are non-trivial tasks to complete with a large dataset with existing tools. The tasks were targeted to test the core features of zenvisage and solicit responses from participants on how zenvisage could complement existing data analytics workflows.

Metrics. We manually recorded how participants used the tool and their answers for each task. We further collected browser activity using screen capture software, system interaction logs, audio of discussions, and asked the participants to complete a survey. Using this data, we collected the following metrics: task completion time, task accuracy, the number of visualizations browsed for each task, the number of edits made on the drawing panel and the query table, and the usability ratings and satisfaction level from the survey results.

Ground Truth. Two expert data analysts prepared the ground truth for each of the tasks in the form of ranked answers. Their inter-rater agreement, measured using Kendall’s Tau rank correlation coefficient, was 0.854. Then, each expert independently rated the answers on their ranked list on a 0 to 5 scale (5 highest). We took the average of the two scores to rate the participants’ answers.

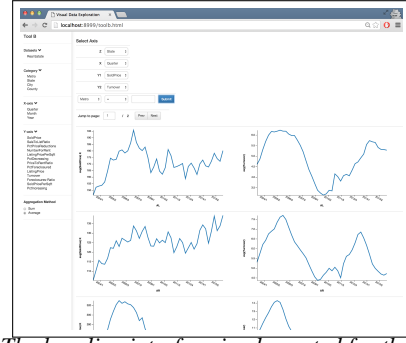


Figure 9: The baseline interface implemented for the user study.

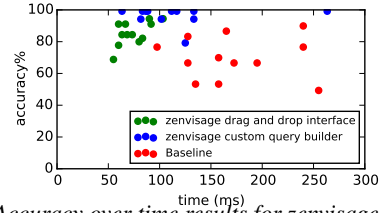


Figure 10: Accuracy over time results for zenvisage and baseline

6.2 Key Findings

Six key findings emerged from the study. We describe them below. We use μ , σ , χ^2 to denote average, standard deviation, and Chi-square test scores, respectively.

Finding 1: zenvisage enabled over 50% and 100% faster task completion times than the baseline for the custom query builder and drag and drop interfaces, respectively. We found that the drag and drop interface had the smallest, and most predictable completion time ($\mu = 74s$, $\sigma = 15.1$), the custom query builder had a higher completion time and a higher variance ($\mu = 115s$, $\sigma = 51.6$), while the baseline had a significantly higher completion time ($\mu = 172.5s$, $\sigma = 50.5$), more than twice the completion time of the drag and drop interface, and almost 50% higher than the custom query builder interface. This is not surprising, given that the baseline tool requires more manual exploration compared to the other two interfaces. Using a one-way between-subjects ANOVA, followed by a post-hoc Tukey’s test [48], we found that the drag and drop interface and the custom query builder interface had statistically significant faster task completion times compared to the baseline interface, with p values of 0.0010 and 0.0069, respectively. The difference between the drag and drop interface and the custom query builder interface was not statistically significant, with a p value of 0.06.

Treatments	Q statistic	p-value	inference
Drag and drop interface vs. Custom query builder	3.3463	0.0605331	insignificant
Drag and drop interface vs. Baseline tool	7.9701	0.0010053	significant ($p < 0.01$)
Custom query builder vs. Baseline tool	4.6238	0.0069276	significant ($p < 0.01$)

Table 19: Tukey’s test on task completion time

Finding 2: zenvisage helped retrieve 20% and 35% more accurate results than the baseline for drag and drop and custom query builder interfaces, respectively. Comparing the participants’ responses against the ground truth responses, we found that participants achieved the lowest accuracy with the baseline tool ($\mu = 69.9\%$, $\sigma = 13.3$). Further, any tasks that required comparing multiple pairs of visualizations or trends ended up having lower accuracy than those that looked for a pattern, such as a peak or an increasing trend, in a single visualization. The low accuracy could be attributed to the fact that the participants selected suboptimal answers before browsing through the entire list of results for better answers. Conversely, zenvisage is able to accept more fine-grained user input, and rank output visualizations. With zenvisage, participants were able to retrieve accurate answers with less effort.

Between the two interfaces in zenvisage, while the drag and drop interface had a faster task completion time, it also had a lower accuracy ($\mu = 85.3\%$, $\sigma = 7.61$). The custom query builder interface had the highest accuracy ($\mu = 96.3\%$, $\sigma = 5.82$). The accuracy was greater than the baseline by over 20% (for drag and drop) and over 35% (for custom builder). Finally, looking at both accuracy and task completion times together in Figure 10, we see that zenvisage helps in “fast-forwarding to desired visualizations” with high accuracy.

Finding 3: The drag and drop interface and the custom query builder complemented each other. The drag and drop interface was intuitive, while the custom builder was useful for complex queries. The majority of participants (8/12) mentioned that the drag and drop and custom query builder interfaces complemented each other in making zenvisage powerful when searching for specific insights. 25 percent (3/12) of the participants (all with backgrounds in machine learning and data mining) preferred using only the custom query builder interface, while just one participant (with no prior experience with query languages or programming languages) found only the drag and drop interface as useful. Participants stated that they would use drag and drop for exploration and simple pattern search and custom query builder for specific / complex queries. One stated: “When I know what I am looking for is exact, I want to use the query table. The other one [drag and drop] is for exploration when you are not so sure.” (P9). Participants liked the simplicity and intuitiveness of the drag and drop interface. One stated: “[The drawing feature] is very easy and intuitive, I can draw... and automatically get the results.” (P2). On a five-point Likert (5 is strongly agree), participants found the drawing feature easy to use ($\mu = 4.45$, $\sigma = 0.674$). Despite the positive feedback, participants did identify limitations, specifically, that the functionality was restricted to identifying trends similar to a single hand-drawn trend.

The majority of the participants (11/12) stated that the custom query builder allowed them to search for complex insights that might be difficult to answer via drag and drop. When asked if the custom query builder was difficult to use compared to the drag-and-drop interface, participants tended to agree ($\mu = 3.73$, $\sigma = 0.866$ on a five-point Likert scale where five is strongly agree). At the same time, participants acknowledged the flexibility and the efficacy supported by the custom query builder interface. One participant explicitly stated why they preferred the custom query builder over the drag and drop interface: “When the query is very complicated, [the drag and drop] has its limitations, so I think I will use the query table most of the times unless I want to get a brief impression.” (P6). Another participant stated that the custom query builder interface “has more functionality and it lets me express what I want to do clearly more than the sketch” (P4). This was consistent with other responses. Participants with a background in machine learning, data mining, and statistics (3/12) preferred using the custom query builder over the drag and drop interface.

Finding 4: zenvisage was more effective (4.27 vs. 2.67 on a five-point Likert scale) than the baseline tool at visual data exploration, and would be a valuable addition to the participants’ analytics workflow. In response to the survey question “I found the tool to be effective in visualizing the data I want to see”, the participants rated zenvisage (the drag and drop interface and the custom query builder interface) higher ($\mu = 4.27$, $\sigma = 0.452$) than the baseline tool ($\mu = 2.67$, $\sigma = 0.890$) on a five-point Likert scale. A participant experienced in Tableau commented: “In Tableau, there is no pattern searching. If I see some pattern in Tableau, such as a decreasing pattern, and I want to see if any other variable is decreasing in that month, I have to go one by one to find this trend. But here I can find this through the query table.” (P10).

Finding 5: zenvisage is a valuable addition to the participants’ analytics workflow, and allows a quicker initial data exploration before performing complex analysis: When asked about using the two tools in their current workflow, 9 of the 12 participants stated that they would use zenvisage in their workflow, whereas two participants stated that they would use our baseline tool ($\chi^2 = 8.22$, $p < 0.01$). When the participants were asked how they would use zenvisage in their workflow, one participant provided a specific scenario: “If I am doing my social science study, and I want to see some specific behavior among users, then I can use tool A [zenvisage] since I can find the trend I am looking for and easily see what users fit into the pattern.” (P7). Other participants mentioned that they would use this tool for finding outliers during data cleaning, common patterns search, and initial understanding of data distributions before running a machine learning task. Participants could not articulate specific reasons for using the baseline tool. As one participant put it, “the existing solutions can already do what tool B [the baseline interface] can do.” (P10). Even those with considerable experience with programming languages such as Python and Matlab argued that zenvisage would take less time and fewer lines of code for basic data exploration tasks. Participants (2/12) who had considerable experience with data mining libraries were asked a follow-up question to write code for a task similar to Table 11 in their favorite language; this task took them multiple orders of magnitudes more time and lines of code compared to ZQL. Example code can be found in the appendix E.

Finding 6: zenvisage can be improved. While the participants looked forward to using custom query builder in their own workflow, a few of them were interested in directly exposing the commonly-used trends/patterns such as outliers, through the drag and drop interface. Some were interested in knowing how they could integrate custom functional primitives (we could not cover it in the tutorial due to time constraints). In order to improve the user experience, participants suggested adding instructions and guidance for new users as part of the interface. Participants also commented on the unrefined look and feel of the tool, as well as the lack of a diverse set of usability related features, such as bookmarking and search history, that are offered in existing systems.

7. EXPRESSIVENESS

In this section, we formally quantify the expressive power of ZQL. To this end, we formulate an algebra, called the *visual exploration algebra*, like relational algebra, with a basic set of operators that we believe all visual exploration languages should be able to express. The operators of our visual exploration algebra operate, at a high-level, on sets of visualizations and is not mired by the data representations of those visualizations, nor the details of how the visualizations are rendered. Instead, the visual exploration algebra is primarily concerned with the different ways in which visualizations can be selected, refined, and compared with each other.

Given a defined measure T for calculating the overall trend of a visualization, a distance metric D for a pair of visualizations, and an algorithm R to identify the most representative visualizations from a set of visualizations, a visual exploration language L is defined to be *visual exploration complete* $VEC_{T,D,R}(L)$ with respect to T , D , and R if it supports all the operators of the visual exploration algebra. These functions T , D , and R (also defined previously) are “exploration functions” without which the resulting algebra would have been unable to manipulate visualizations in the way we need for data exploration. Unlike relational algebra, which does not have any “black box” functions, visual exploration algebra requires these functions for operating on visualizations effectively. That said, these three functions are flexible and configurable and up

to the user to define (or left as system defaults). Next, we formally define the visual exploration algebra operators and prove that ZQL is visual exploration complete.

7.1 Basic Notation

Assume we are given a k -ary relation \mathcal{R} , which has attributes (A_1, A_2, \dots, A_k) . Let \mathcal{X} be the unary relation with attribute X whose values are the names of the attributes in \mathcal{R} that can appear on the x -axis. If the x -axis attributes are not specified by the user for relation \mathcal{R} , the default behavior is to include all attributes in \mathcal{R} : $\{A_1, \dots, A_k\}$. Let \mathcal{Y} be defined similarly with Y for attributes that appear on the y -axis. Given \mathcal{R} , \mathcal{X} , and \mathcal{Y} , we define \mathcal{V} , the *visual universe*, as follows: $\mathcal{V} = v(\mathcal{R}) = \mathcal{X} \times \mathcal{Y} \times \left(\times_{i=1}^k \pi_{A_i}(\mathcal{R}) \cup \{*\} \right)$ where π is the projection operator from relational algebra and $*$ is a special wildcard symbol, used to denote all values of an attribute. Table 20 shows an example of what a sample \mathcal{R} and corresponding \mathcal{X} , \mathcal{Y} , and \mathcal{V} would look like. At a high level, the visual universe specifies all subsets of data that may be of interest, along with the intended attributes to be visualized. Unlike relational algebra, visual exploration algebra mixes schema and data elements, but in a special way in order to operate on a collection of visualizations.

Any subset relation $V \subseteq \mathcal{V}$ is called a *visual group*, and any $k+2$ -tuple from \mathcal{V} is called a *visual source*. The last k portions of \mathcal{V} are called the *data source* of the visual source. At a high level, a visual source represents a visualization which can be rendered from a selected data source, and a set of visual sources is a visual group. The X and Y attributes of the visual source determine the x - and y - axes, and the selection on the data source is determined by attributes A_1, \dots, A_k . If an attribute has the wildcard symbol $*$ as its value, no subselection is performed on that attribute for the data source. For example, the third row of Table 20d is a visual source that represents the visualization with year as the x -axis and sales as the y -axis for chair products. Since the value of location is $*$, all locations are considered valid or pertinent for the data source. In relational algebra, the data source for the third row can be written as $\sigma_{\text{product}=\text{chair}}(\mathcal{R})$. The $*$ symbol therefore attempts to emulate the lack of presence of a selection condition on that attribute in the σ operator of the relational algebra. Readers familiar with OLAP will notice the similarity between the use of the $*$ here and the GROUPING SETS functionality in SQL.

Note that infinitely many visualizations can be produced from a single visual source, due to different granularities of binning, aggregation functions, and types of visualizations that can be constructed, since a visualization generation engine can use a visualization rendering grammar like *ggplot* [50] that provides that functionality. Our focus in defining the visual exploration algebra is to specify the inputs to a visualization and attributes of interest as opposed to the aesthetic aspects and encodings. Thus, for our discussion, we assume that each visual source maps to a singular visualization. Even if the details of the encoding and aesthetics are not provided, standard rules may be applied for this mapping as alluded earlier [28, 47]. Furthermore, a visual source does not specify the data representation of the underlying data source; therefore the expressive power of visual exploration algebra is not tied to any specific backend data storage model. The astute reader will have noticed that the format for a visual source looks fairly similar to the visual components of ZQL; this is no accident. In fact, we will use the visual components of ZQL as a proxy to visual sources when proving that ZQL is visual exploration complete.

7.2 Ordered Bag Semantics

In visual exploration algebra, relations have bag semantics. However, since users want to see the most relevant visualizations first,

ordering is critical. So, we adapt the operators from relational algebra to preserve ordering information.

Thus, we operate on *ordered bags*. We describe the details of how to operate on ordered bags below. We use R, S to denote the ordered bags. We also use the notation $R = [t_1, \dots, t_n]$ to refer to an ordered bag, where t_i are the tuples.

The first operator that we define is an indexing operator, much like indexing in arrays. $R[i]$ refers to the i th tuple within R , and $R[i : j]$ refers to the ordered bag corresponding to the list of tuples from the i th to the j th tuple, both inclusive. In the notation $[i : j]$ if either one of i or j is omitted, then it is assumed to be 1 for i , and n for j , where n is the total number of tuples.

Next, we define a union operator \cup . $R \cup S$ refers to the concatenation of the two ordered bags R and S . We define the union operation first because it will come in handy for subsequent operations. If one of R or S is empty, then the result of the union is simply the other relation.

We define the σ operator like in relational algebra, via a recursive definition:

$$\sigma_\theta(R) = \sigma_\theta([R[1]]) \cup \sigma_\theta(R[2 :])$$

where σ_θ when applied to an ordered bag with a single tuple $([t])$ behaves exactly like in the relational algebra case, returning the same ordered bag $([t])$ if the condition is satisfied, and the empty ordered bag if the condition is not satisfied. The π operator for projection is defined similarly to σ in the equation above, with the π operator on an ordered bag with a single tuple simply removing the irrelevant attributes from that tuple, like in the relational algebra setting.

Then, we define the \setminus operator, for ordered bag difference. Here, the set difference operator operates on every tuple in the first ordered bag and removes it if it finds it in the second ordered bag. Thus:

$$R \setminus S = ([R[1]] \setminus S) \cup (R[2 :] \setminus S)$$

where $[t] \setminus S$ is defined like in relational algebra, returning $[t]$ if $[t]$ is not in S , and $[]$ otherwise.

Now, we can define the duplicate elimination operator as follows:

$$\delta(R) = [R[1]] \cup (R[2 :] \setminus [R[1]])$$

Thus, the duplication elimination operator preserves ordering, while maintaining the first copy of each tuple at the first position that it was found in the ordered bag.

Lastly, we have the cross product operator, as follows:

$$R \times S = ([R[1]] \times S) \cup (R[2 :] \times S)$$

where further we have

$$[t] \times S = ([t] \times [S[1]]) \cup ([t] \times S[2 :])$$

where $[t] \times [u]$ creates an ordered bag with the result of the cross product as defined in relational algebra. Given these semantics for ordered bags, we can develop the visual exploration algebra.

7.3 Exploration Functions

Earlier, we mentioned that a visual exploration algebra is visual exploration complete with respect to three exploration functions T, R , and D . Here we define the type of these exploration functions and describe them in more detail.

The function $T : \mathcal{V} \rightarrow \mathcal{R}$ returns a real number given a visual source. This function can be used to assess whether a trend: defined by the visualization corresponding to a specific visual source,

Table 20: An example relation \mathcal{R} and its resultant \mathcal{X} , \mathcal{Y} , and \mathcal{V} .

year	month	product	location	sales	profit					X	Y			X	Y	X	Y	year	month	product	location	sales	profit
2016	4	chair	US	623,000	314,000					year	sales			year	sales	year	sales	*	*	*	*	*	*
2016	3	chair	US	789,000	410,000					month	profit			year	profit	year	profit	*	*	*	*	*	*
2016	4	table	US	258,000	169,000									year	sales	year	sales	*	*	chair	*	*	*
2016	4	chair	UK	130,000	63,000									year	sales	year	sales	*	*	chair	US	*	*

(a) Example \mathcal{R}

(b) \mathcal{X}

(c) \mathcal{Y}

(d) \mathcal{V} for \mathcal{R}

is “increasing”, or “decreasing”, or satisfies some other fixed property. Many such T can be defined and used within the visual exploration algebra.

The function $D : \mathcal{V} \times \mathcal{V} \rightarrow \mathcal{R}$ returns a real number given a pair of visual sources. This function can be used to compare pairs of visualizations (corresponding to the visual sources) with respect to each other. The most natural way to define D is via some notion of distance, e.g., Earth Mover’s or Euclidian distance, but once again, can be provided by the user or assumed as a fixed black box.

The function $R : \mathcal{V}^n \rightarrow \mathcal{R}^n$ given a list of visual sources returns a list of values, one corresponding to each of the visual sources. This function can take in an arbitrary long list of visual sources, and returns a score of for each one. This function’s intended use is for ascertaining representativeness, i.e., how representative do we believe the visualization corresponding to a specific visual source to be. The function could equally well be used for ascertaining outlieriness, or some other global property that requires consideration of the entire set of visualizations with respect to each other.

7.4 Visual Exploration Algebra Operators

Similar to how operators in ordered bag algebra operate on and result in ordered bags, operators in visual exploration algebra operate on and result in visual groups. Many of the symbols for operators in visual exploration algebra are also derived from ordered bag algebra, with some differences. To differentiate, operators in visual exploration algebra are superscripted with a v (e.g., σ^v , τ^v). The unary operators for visual exploration algebra include (i) σ^v for selection, (ii) τ^v for sorting a visual group based on trend-estimating function T , (iii) μ^v for limiting the number of visual sources in a visual group, (iv) δ^v for duplicate visual source removal, and (v) ζ^v for finding the most representative visual sources of a visual group based on algorithm R . The binary operators include (i) \cup^v for union, (ii) \setminus^v for difference, (iii) β^v for replacing the attribute values of the visual sources in one visual group’s with another’s, (iv) ϕ^v to reorder the first visual group based on the visual sources’ distances to the visual sources of another visual group based on metric D , and (v) η^v to reorder the visual sources in a visual group based on their distance to a reference visual source from a singleton visual group based on D . These operators are described below, and listed in Table 21:

Unary Operators. $\sigma_\theta^v(V)$: σ^v selects a visual group from V based on selection criteria θ , like ordered bag algebra. However, σ^v has a more restricted θ ; while \vee and \wedge may still be used, only the binary comparison operators $=$ and \neq are allowed. As an example, $\sigma_\theta^v(V)$ where $\theta = (X='year' \wedge Y='sales' \wedge year=* \wedge month=* \wedge product \neq * \wedge location='US' \wedge sales=* \wedge profit=*)$ from Table 22 on \mathcal{V} from Table 20 would result in the visual group of time vs. sales visualizations for different products in the US.

Note that the product is specifically set to not equal $*$ so that the resulting visual group will include all products. On the other hand, the location is explicitly set to be equal to US. The other attributes, e.g., sales, profit, year, month are set to equal $*$: this implies that the visual groups are not employing any additional constraints on those attributes. (This may be useful, for example when those attributes are not relevant for the current visualization or set of visualizations.) As mentioned before, visual groups have the semantics

of ordered bags. Thus, σ^v operates on one tuple at a time of the given V in the order they appear in V , and the result is in the same order the tuples are operated on.

$\tau_{F(T)}^v(V)$: τ^v returns visual group sorted in an increasing order based on applying $F(T)$ on each visual source in V , where $F(T)$ is a procedure that uses function T . For example, $\tau_{-T}^v(V)$ might return the visualizations in V sorted in decreasing order of estimated slope. This operator is not present in the ordered bag semantics, but may be relevant when we want to reorder the ordered bag using a different criterion.

$\mu_k^v(V)$: μ^v returns the first k visual sources of V ordered in the same way they were in V . μ^v is equivalent to the LIMIT statement in SQL. μ^v is often used in conjunction with τ^v to retrieve the top- k visualizations with gradient slopes (e.g. $\mu_k^v(\tau_{-T}^v(V))$). When instead of a number k , the subscript to μ^v is actually $[a : b]$, then the items of V that are between positions a and b in V are returned. Thus μ^v offers identical functionality to the $[a : b]$ in ordered bag algebra, with the convenient functionality of getting the top k results by just having one number as the subscript. Instead of using μ^v , visual exploration algebra also supports the use of the syntax $V[i]$ to refer to the i th visual source in V , and $V[a : b]$ to refer to the ordered bag of visual sources from positions a to b .

$\delta^v(V)$: δ^v returns the visual sources in V with the duplicates removed, in the order of their first appearance. Thus, δ^v is defined identically to ordered bag algebra.

$\zeta_{R,k}^v(V)$: ζ^v returns the k -most representative visual sources from V based on representative-finding algorithm R . The returned results may be in any order. Unlike ordered bag algebra, which does not have this functionality, visual exploration algebra has a special purpose operator that uses the black box function R to return representative visual sources, from among all of V : implicitly, this black box function can compare all of V to itself.

Binary Operators. $V \cup^v U \mid V \setminus^v U$: Returns the union / difference of V and U . These operations are just like the corresponding operations in ordered bag algebra.

$\beta_A^v(V, U)$: β^v returns a visual group in which values of attribute A in V are replaced with the values of A in U . Formally, assuming A_i is the i th attribute of V and V has n total attributes: $\beta_{A_i}^v(V, U) = \pi_{A_1, \dots, A_{i-1}, A_{i+1}, \dots, A_n}(V) \times \pi_{A_i}(U)$. This can be useful for when the user would like to change an axis: $\beta_X(V, \sigma_{X=year}^v(V))$ will change the visual sources in V to have year as their x-axis. β^v can also be used to combine multiple dimensions as well. If we assume that V has multiple Y values, we can do $\beta_X^v(V, \sigma_{X \neq *}^v(V))$ to have the visual sources in V vary over both X and Y . This operator allows us to start with a set of visualizations and then “pivot” to focus on a different attribute, e.g., start with sales over time visualizations and pivot to look at profit. Thus, the operator allows us to transform the space of visual sources.

$\phi_{F(D)}^v(V, U)$: ϕ^v sorts the visual sources in V in increasing order based on their distances to the visual sources in U . If we denote $R[i]$ to return the i th tuple of relation R , then ϕ^v computes $F(D)(V[i], U[i]) \forall i \in \{1, \dots, |V|\}$, and returns a reordered V based on these values, where F is a procedure that uses D . The operation is undefined if $|V| \neq |U|$. This operator supports comparative

Operator	Name	Derived from Bag Algebra	Meaning	Unary/Binary
σ^v	Selection	Yes	Subselects visual sources	Unary
τ^v	Sort	No	Sorts visual sources in increasing order	Unary
μ^v	Limit	Yes	Returns first k visual sources	Unary
δ^v	Dedup	Yes	Removes duplicate visual sources	Unary
ζ^v	Representative	No	Selects k representative visual sources	Unary
\cup^v/\setminus^v	Union/Diff	Yes	Returns the union of/differences between two visual groups	Binary
β^v	Swap	No	Returns a visual group in which values of an attribute in one visual group is replaced with values of the same attribute in another visual group	Binary
ϕ^v	Dist	No	Sorts a visual group based on pairwise distance to another visual group	Binary
η^v	Find	No	Sorts a visual group in increasing order based on their distances to a single reference visual source	Binary

Table 21: Visual Exploration Algebra Operators

X	Y	year	month	product	location	sales	profit
year	sales	*	*	chair	US	*	*
year	sales	*	*	table	US	*	*
year	sales	*	*	stapler	US	*	*
year	sales	*	*	printer	US	*	*
				...			

Table 22: Results of performing unary operators on \mathcal{V} from Table 20: $\sigma_\theta^v(\mathcal{V})$ where $\theta = (X='year' \wedge Y='sales' \wedge year=* \wedge month=* \wedge product \neq * \wedge location='US' \wedge sales=* \wedge profit=*)$

queries similar to Table 12 where we find the x- and y- axes for which the visualizations are most different: $\phi_{-D,X,Y}^v(V,U)$. This operator is analogous to τ^v , except that it operates on pairs of visual sources at a time.

$\eta_{F(D)}^v(V,U)$: η^v sorts the visual sources in V in increasing order based on their distances to a single reference visual source in singleton visual group U . Thus, $U = [t]$. η^v computes $F(D)(V[i], U[1]) \forall i \in \{1, \dots, |V|\}$, and returns a reordered V based on these values, where $F(D)$ is a procedure that uses D . If U has more than one visual source, the operation is undefined. η^v is useful for queries in which the user would like to find the top- k most similar visualizations to a reference: $\mu_k^v(\eta_{F(D)}^v(V,U))$, where V is the set of candidates and U contains the reference. Once again, this operator is similar to τ^v , except that it operates on the results of the comparison of individual visual sources to a specific visual source.

7.5 Proof of Visual Exploration Completeness

Given the visual exploration algebra operators as defined previously, we can show that ZQL is visual exploration complete. A key lemma to this proof is the idea that the visual components of ZQL serve as an appropriate proxy for the visual groups in visual exploration algebra.

LEMMA 7.1. *The visual components of ZQL have at least as much expressive power as the visual groups in visual exploration algebra.*

PROOF. A visual group V , with n visual sources, is a relation with $k+2$ columns and n rows, where k is the number of attributes in the original relation. We show that for any visual group V , we can come up with a ZQL query q which can produce a visual component that represents the same set of visualizations as V .

Name	X	Y	Z1	...	Zk
f1	$\pi_X(V[1])$	$\pi_Y(V[1])$	$E_{1,1}$...	$E_{1,k}$
...
fn	$\pi_X(V[n])$	$\pi_Y(V[n])$	$E_{n,1}$...	$E_{n,k}$
*fn+1=f1+...+fn					

Table 23: ZQL query q which produces a visual component equal in expressivity to visual group V .

Query q has the format given by Table 23, where $V[i]$ denotes the i th tuple of relation V and:

$$E_{i,j} = \begin{cases} "" & \text{if } \pi_{A_j}(V[i]) = * \\ A_j \cdot \pi_{A_j}(V[i]) & \text{otherwise} \end{cases}$$

Here, A_j refers to the j th attribute of the original relation. The i th visual source of V is represented with the \mathbf{fi} from q . The X and Y values come directly from the visual source using projection. For the Z_j column, if the A_j attribute of visual source has any value other than $*$, we must filter the data based on that value, so $E_{i,j} = A_j \cdot \pi_{A_j}(V[i])$. However, if the A_j attribute is equal to $*$, then the corresponding element in \mathbf{fi} is left blank, signaling no filtering based on that attribute.

After, we have defined a visual component \mathbf{fi} for each i th visual source in V , we take the sum across all these visual components as defined in Appendix A.5, and the resulting $\mathbf{fn}+1$ becomes equal to the visual group V . \square

LEMMA 7.2. $\tau_{F(T)}^v(V)$ is expressible in ZQL for all valid constraints functions F of T and visual groups V .

PROOF. Assume $\mathbf{f1}$ is the visual component which represents V . Query q given by Table 24 produces visual component $\mathbf{f3}$ which expresses $\tau_{F(T)}^v(V)$. \square

LEMMA 7.3. $\mu_{[a:b]}^v(V)$ is expressible in ZQL for all valid intervals $a : b$ and visual groups V .

PROOF. Assume $\mathbf{f1}$ is the visual component which represents V . Query q given by Table 25 produces visual component $\mathbf{f2}$ which expresses $\mu_{[a:b]}^v(V)$. \square

LEMMA 7.4. $\zeta_{R,j}^v(V)$ is expressible in ZQL for all valid numbers j and visual groups V .

PROOF. Assume $\mathbf{f1}$ is the visual component which represents V and R' is the corresponding representative-finding algorithm in ZQL. Query q given by Table 26 produces visual component $\mathbf{f3}$ which expresses $\mu_{[a:b]}^v(V)$. \square

LEMMA 7.5. $V \cup^v U$ is expressible in ZQL for all valid visual groups V and U .

PROOF. Assume $\mathbf{f1}$ is the visual component which represents V and $\mathbf{f2}$ represents U . Query q given by Table 28 produces visual component $\mathbf{f3}$ which expresses $V \cup^v U$. \square

LEMMA 7.6. $\eta_{F(D)}^v(V,U)$ is expressible in ZQL for all valid functionals F of D and visual groups V and singleton visual groups U .

Name	X	Y	Z1	...	Zk	Process
f1	-	-	-	...	-	
f2=f1	x1 <- -	y1 <- -	v1 <- A1.-	...	vk <- Ak.-	x2, y2, u1, ..., uk <- argmin _{x1,y1,v1,...,vk} [k = ∞]F(T)(f2)
*f3	x2	y2	u1	...	uk	

Table 24: ZQL query q which expresses $\tau_{F(T)}^v(V)$.

Name	X	Y	Z1	...	Zk	Process
f1	-	-	-	...	-	
*f2=f1[a:b]						

Table 25: ZQL query q which expresses $\mu_{[a:b]}^v(V)$.

Name	X	Y	Z1	...	Zk	Process
f1	-	-	-	...	-	
f2=f1	x1 <- -	y1 <- -	v1 <- A1.-	...	vk <- Ak.-	x2, y2, u1, ..., uk <- R(j,x1,y1,v1,...,vk,f2)
*f3	x2	y2	u1	...	uk	

Table 26: ZQL query q which expresses $\zeta_{R,j}^v(V)$.

Name	X	Y	Z1	...	Zk	Process
f1	-	-	-	...	-	
f2	-	-	-	...	-	
*f3=f1+f2						

Table 27: ZQL query q which expresses $V \cup^v U$.

Name	X	Y	Z1	...	Zk	Process
f1	-	-	-	...	-	
f2	-	-	-	...	-	
f3=f1	x1 <- -	y1 <- -	v1 <- A1.-	...	vk <- Ak.-	x2, y2, u1, ..., uk <- argmin _{x1,y1,v1,...,vk} [k = ∞]F(D)(f3,f2)
*f4	x2	y2	u1	...	uk	

Table 28: ZQL query q which expresses $\eta_{F(D)}^v(V, U)$.

PROOF. Assume f1 is the visual component which represents V and f2 represents U . Query q given by Table 28 produces visual component f4 which expresses $\eta_{F(D)}^v(V, U)$. \square

THEOREM 7.7. *Given well-defined functions T , D , and R , ZQL is visual exploration complete with respect to T , D , and R .*

We have shown that a visual component is capable of representing a visual group, and ZQL has the capability to express the operators in visual exploration algebra (we omit the full proofs for some of the operators due to them being similar in nature). As such, Theorem 7.7 holds.

Although we have come up with a formalized algebra to measure the expressiveness of ZQL, ZQL is actually more expressive than visual exploration algebra. For example, ZQL allows the user to nest multiple levels of iteration in the Process column as in Table 13. Nevertheless, visual exploration algebra serves as a useful minimum metric for determining the expressiveness of visual exploration languages. Other visual analytics tools like Tableau are capable of expressing the selection operator σ^v in visual exploration algebra, but they are incapable of expressing the other operators which compare and filter visualizations based on summarization functions T , D , and R . General purpose programming languages with analytics libraries such as Python and Scikit-learn [40] are visual exploration complete since they are Turing-complete, but ZQL's declarative syntax strikes a novel balance between simplicity and expressiveness which allows even non-programmers to become data analysts as we saw in Section 6.

8. RELATED WORK

In this section, we discuss related prior work in a number of areas. To aid our discussion, in Figure 11, we depict our abstract conceptualization of the three stages of data analytics; our specific target, with zenvisage, is not on the first or third stages, which are better handled by other tools, but instead on the middle stage, which is also the focus of current visual analytics tools. Many of these tools target both programmers and non-programmers (like we do). **Visual Analytics Tools:** Visual analytics tools, such as ShowMe, Spotfire, and Tableau [47, 37, 11] have recently gained in popularity. While these tools, along with similar tools from the database

community [22, 34, 35, 29] support the selection and generation of visualizations, via interactions, they rely on cumbersome manual examination for identifying patterns or insights.

OLAP Browsing Tools: There has been some limited work on interactive browsing of data cubes, e.g., [42, 43]. While we may be able to reuse some of the metrics from that line of work, the work focuses on suggestions for raw aggregates (cells) to examine that are informative given past browsing, or those that show a generalization or explanation of a specific cell, and not on two (or more) dimensional visualizations, or the full data exploration capabilities (e.g., searching for trends, patterns, outliers) provided by ZQL.



Figure 11: The stages of data analytics: the shaded box is our focus

Data Mining Languages: There has been some limited work in data mining query languages, all from the early 90s, on association rule mining (DMQL [24], MSQL [27]), or on storing and retrieving models on data (OLE DB [38]), as opposed to a visual data exploration language aimed at identifying visual trends or insights.

Statistical Packages and Programming Libraries: Statistical analysis tools [3, 4, 5, 7] support complex data mining primitives, but require extensive knowledge of the underlying algorithms and parametrization, limiting their use to experts, e.g., a user can select to use decision trees, with a certain depth and splitting criteria on their dataset using any of these tools. Programming libraries such as Weka [25] and Scikit-learn [40] support embedding machine learning and data mining within programs. The statistical analysis and programming libraries falls under the third box of Figure 11, representing the third stage of data analytics, following data exploration. Indeed, complex statistical tasks such as classification, regression, and dimensionality reduction are hugely important, but require programming ability or an understanding of machine learning and statistics to be useful. zenvisage is instead intended to be a substantial generalization over existing data exploration tools (second box in Figure 11), aimed at identifying the patterns and trends that hold in the data, by non-programmers, which can be then used to train a machine learning algorithm later on.

Visualization Recommendation Tools: There has been some recent work on building systems that recommend visualizations. Voy-

ager [28] recommends visualizations based on aesthetic properties of the visualizations, as opposed to queries. SeeDB [49] recommends visualizations that best display the difference between two sets of data. SeeDB and Voyager can be seen to be special cases of zenvisage. The optimization techniques outlined are a generalization of the techniques described in SeeDB; while the techniques in SeeDB are special-cased to the query considered (a simple comparison), here, our goal is to support and optimize all ZQL queries.

Multi-Query Optimization: There has been a lot of work on Multi-Query Optimization (MQO), both classic, e.g., [44, 45, 41], and recent work, e.g., [20, 26, 30, 21]. Overall, the approach adopted is to batch queries, decompose into operators, and build “meta”-query plans that process multiple queries at once, with sharing at the level of scans, or at the level of higher level operators (either via simultaneous pipelining or a true global query plan [26]). Unlike these techniques which require significant modifications to the underlying database engine—indeed, some of these systems do not even provide full cost-based optimization and only support hand-tuned plans [20], in this paper, we adopted two syntactic rewriting techniques that operate outside of any relational database as a back-end without requiring any modification, and can thus seamlessly leverage improvements to the database. Our third optimization is tailored to the ZQL setting and does not apply more broadly.

Anomaly Discovery: Anomaly detection is a well-studied topic [16, 10, 39]. Our goal in that zenvisage is expected to be interactive, especially on large datasets; most work in anomaly detection focuses on accuracy at the cost of latency and is typically a batch operation. In our case, since interactivity is of the essence, and requests can come at any time, the emphasis is on scalable on-the-fly data processing aspects.

Time Series Similarity and Indexing: There has been some work on indexing of time series data, e.g., [33, 23, 15, 32, 13, 19, 31]; for the attributes that are queried frequently, we plan to reuse these techniques for similarity search. For other attributes, indexing and maintaining all trends is impossible, since the number of trends grows exponentially with the number of indexed attributes.

9. CONCLUSION

We propose zenvisage, a tool for effortless specification and visualization of interesting patterns and insights from large datasets. The bulk of our paper focused on the formal specification of ZQL, our query language. We described how ZQL captures a range of interesting and useful scenarios, and we showed through visual exploration algebra that ZQL is powerful enough to capture all of the typical data exploration queries users wish to issue. zenvisage allows both novice and expert users to effectively perform visual exploration tasks, as shown by our user study. In addition, we show that our optimizations for ZQL translation lead to several orders of performance improvement on real and synthetic datasets. While our work is a promising first step towards simplifying and improving visual data exploration, much more work remains to be done in further improving optimization, learning from user feedback, and providing better recommendations.

10. REFERENCES

- [1] Airline dataset. [Online; accessed 30-Oct-2015].
- [2] Census income dataset. [Online; accessed 30-Oct-2015].
- [3] Knime. [Online; accessed 30-Oct-2015].
- [4] Rapidminer dataset. [Online; accessed 30-Oct-2015].
- [5] Sas. [Online; accessed 30-Oct-2015].
- [6] Spotfire, <http://spotfire.com>. [Online; accessed 17-Aug-2015].
- [7] Spss. [Online; accessed 30-Oct-2015].
- [8] Tableau public. [Online; accessed 3-March-2014].
- [9] Zillow real estate data. [Online; accessed 1-Feb-2016].
- [10] M. Agyemang, K. Barker, and R. Alhajj. A comprehensive survey of numeric and symbolic outlier mining techniques. *Intell. Data Anal.*, 10(6):521–538, Dec. 2006.
- [11] C. Ahlberg. Spotfire: An information exploration environment. *SIGMOD Rec.*, 25(4):25–29, Dec. 1996.
- [12] K. S. Bordens and B. B. Abbott. *Research design and methods: A process approach*. McGraw-Hill, 2002.
- [13] K. Chakrabarti, E. Keogh, S. Mehrotra, and M. Pazzani. Locally adaptive dimensionality reduction for indexing large time series databases. *ACM Trans. Database Syst.*, 27(2):188–228, June 2002.
- [14] S. Chambi, D. Lemire, O. Kaser, and R. Godin. Better bitmap performance with roaring bitmaps. *Software: Practice and Experience*, 2015.
- [15] K.-P. Chan and A.-C. Fu. Efficient time series matching by wavelets. In *Data Engineering, 1999. Proceedings., 15th International Conference on*, pages 126–133, Mar 1999.
- [16] V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection: A survey. *ACM Comput. Surv.*, 41(3):15:1–15:58, July 2009.
- [17] J. W. Creswell and V. L. P. Clark. *Designing and conducting mixed methods research*. 2007.
- [18] N. Elmqvist, J. Stasko, and P. Tsigas. Datameadow: a visual canvas for analysis of large-scale multivariate data. *Information visualization*, 7(1):18–33, 2008.
- [19] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. Fast subsequence matching in time-series databases. *SIGMOD Rec.*, 23(2):419–429, May 1994.
- [20] G. Giannikis et al. Workload optimization using shareddb. In *SIGMOD*, pages 1045–1048. ACM, 2013.
- [21] G. Giannikis et al. Shared workload optimization. *Proceedings of the VLDB Endowment*, 7(6):429–440, 2014.
- [22] H. Gonzalez et al. Google fusion tables: web-centered data management and collaboration. In *SIGMOD Conference*, pages 1061–1066, 2010.
- [23] D. Gunopulos and G. Das. Time series similarity measures and time series indexing (abstract only). *SIGMOD Rec.*, 30(2):624–, May 2001.
- [24] J. Han et al. Dmql: A data mining query language for relational databases. In *Proc. 1996 SIGMOD*, volume 96, pages 27–34, 1996.
- [25] G. Holmes, A. Donkin, and I. H. Witten. Weka: A machine learning workbench. In *Conf. on Intelligent Information Systems '94*, pages 357–361. IEEE, 1994.
- [26] I. Psaroudakis et al. Sharing data and work across concurrent analytical queries. *VLDB*, 6(9):637–648, 2013.
- [27] T. Imielinski and A. Virmani. A query language for database mining. *Data Mining and Knowledge Discovery*, 3(4):373–408, 2000.
- [28] K. Wongsuphasawat et al. Voyager: Exploratory analysis via faceted browsing of visualization recommendations. *IEEE TVCG*, 2015.
- [29] S. Kandel et al. Profiler: integrated statistical analysis and visualization for data quality assessment. In *AVI*, pages 547–554, 2012.
- [30] A. Kementsietsidis et al. Scalable multi-query optimization for exploratory queries over federated scientific databases. *PVLDB*, 1(1):16–27, 2008.
- [31] E. Keogh. A decade of progress in indexing and mining large

- time series databases. In *Proceedings of the 32Nd International Conference on Very Large Data Bases, VLDB '06*, pages 1268–1268. VLDB Endowment, 2006.
- [32] E. Keogh, K. Chakrabarti, M. Pazzani, and S. Mehrotra. Dimensionality reduction for fast similarity search in large time series databases. *Knowledge and Information Systems*, 3(3):263–286, 2001.
- [33] E. Keogh, K. Chakrabarti, M. Pazzani, and S. Mehrotra. Locally adaptive dimensionality reduction for indexing large time series databases. *SIGMOD Rec.*, 30(2):151–162, May 2001.
- [34] A. Key, B. Howe, D. Perry, and C. Aragon. Vizdeck: Self-organizing dashboards for visual analytics. *SIGMOD '12*, pages 681–684, 2012.
- [35] M. Livny et al. Devise: Integrated querying and visualization of large datasets. In *SIGMOD Conference*, pages 301–312, 1997.
- [36] J. Mackinlay. Automating the design of graphical presentations of relational information. *ACM Trans. Graph.*, 5(2):110–141, Apr. 1986.
- [37] J. D. Mackinlay et al. Show me: Automatic presentation for visual analysis. *IEEE Trans. Vis. Comput. Graph.*, 13(6):1137–1144, 2007.
- [38] A. Netz et al. Integrating data mining with sql databases: Ole db for data mining. In *ICDE'01*, pages 379–387. IEEE, 2001.
- [39] A. Patcha and J.-M. Park. An overview of anomaly detection techniques: Existing solutions and latest technological trends. *Comput. Netw.*, 51(12):3448–3470, Aug. 2007.
- [40] Pedregosa et al. Scikit-learn: Machine learning in python. *The Journal of Machine Learning Research*, 12:2825–2830, 2011.
- [41] P. Roy et al. Efficient and extensible algorithms for multi query optimization. In *ACM SIGMOD Record*, volume 29, pages 249–260. ACM, 2000.
- [42] S. Sarawagi. Explaining differences in multidimensional aggregates. In *VLDB*, pages 42–53, 1999.
- [43] G. Sathe and S. Sarawagi. Intelligent rollups in multidimensional olap data. In *VLDB*, pages 531–540, 2001.
- [44] T. K. Sellis. Multiple-query optimization. *ACM TODS*, 13(1):23–52, 1988.
- [45] K. Shim et al. Improvements on a heuristic algorithm for multiple-query optimization. *Data & Knowledge Engineering*, 12(2):197–222, 1994.
- [46] M. Slee, A. Agarwal, and M. Kwiatkowski. Thrift: Scalable cross-language services implementation. *Facebook White Paper*, 5(8), 2007.
- [47] C. Stolte et al. Polaris: a system for query, analysis, and visualization of multidimensional databases. *Commun. ACM*, 51(11):75–84, 2008.
- [48] J. W. Tukey. Comparing individual means in the analysis of variance. *Biometrics*, pages 99–114, 1949.
- [49] M. Vartak, S. Rahman, S. Madden, A. Parameswaran, and N. Polyzotis. Seedb: Efficient data-driven visualization recommendations to support visual analytics. *Proc. VLDB Endow.*, 8(13), Sept. 2015.
- [50] H. Wickham. ggplot: An implementation of the grammar of graphics. *R package version 0.4. 0*, 2006.
- [51] L. Wilkinson. *The grammar of graphics*. Springer Science & Business Media, 2006.
- [52] M. M. Zloof. Query-by-example: A data base language. *IBM Systems Journal*, 16(4):324–343, 1977.

APPENDIX

Here, we provide additional details on X, Y, Z, Constraints, and Process columns (Appendix A), the pseudocode for the naive translation for query execution (Appendix B), four real-world complex examples (Appendix C), performance comparison between the PostgreSQL and Roaring Bitmap databases (Appendix D), and finally we provide a participant’s python code implementation (Appendix E) for a ZQL task to support the finding 5 in the user-study (Section 6).

A. QUERY LANGUAGE FORMALIZATION: ADDITIONAL DETAILS

In this section, we present some additional details on our formalization that we did not cover in the main body of the paper.

A.1 Additional Details on X and Y Columns

In addition to using a single attribute for an X or Y column, ZQL also allows the use of the Polaris table algebra [8] in the X and Y columns to to arbitrarily compose multiple attributes into a single attribute; all three operators are supported: $+$, \times , $/$. Table 29 shows an example of using the $+$ operator to visualize both profits and sales on a single y-axis. Note that this is different from the example given in Table 4, which generates two visualizations, as opposed to a single visualization. An example using both table algebra and sets is given in Table 30, which uses the \times operator to return the set of visualizations which measures the sales for the Cartesian product of ‘product’ and one of ‘county’, ‘state’, and ‘country’.

Name	X	Y	Constraints
*f1	‘product’	‘profit’ + ‘sales’	location=‘US’

Table 29: A ZQL query for a visualization which depicts both profits and sales on the y-axis for products in the US.

Name	X	Y
*f1	‘product’ \times {x1 in {‘county’, ‘state’, ‘country’}}	‘sales’

Table 30: A ZQL query for the set of visualizations which measures the sales for one of (‘product’, ‘county’), (‘product’, ‘state’), and (‘product’, ‘country’).

A.2 Additional Details on the Z Column

ZQL also allows the iteration over attributes in the Z column as shown in Table 31. The result of this query is the set of all sales over time visualizations for every possible slice in every dimension except ‘time’ and ‘sales’. Since both attribute and attribute value can vary in this case, we need separate variables for each component, and the full attribute name, value pair (z1.v1) must be specified. Note that the resulting set of visualizations comes from the Cartesian product of possible attribute and attribute value pairs. The first * symbol refers to all possible attributes, while the second * symbol refers to all possible attribute values given an attribute. If the user wishes to specify specific subsets of attribute values for attributes, she must name them individually. An example of this is given in Table 32 where the z1 and v1 iterate over the pairs {(‘product’, ‘chair’), (‘product’, ‘desk’), (‘location’, ‘US’)}.

Name	X	Y	Z
f1	‘year’	‘sales’	z1.v1 <- (\ {‘year’, ‘sales’}).*

Table 31: A ZQL query which returns the set of sales over year visualizations for each attribute that is not time or sales.

Finally, it is possible to have multiple Z columns, named Z2, Z3, ... in ZQL, increasing the number of ways of simultaneously slicing through the data—these Z columns would be akin to *layers* in Polaris. These multiple Z columns enable us to increase the space

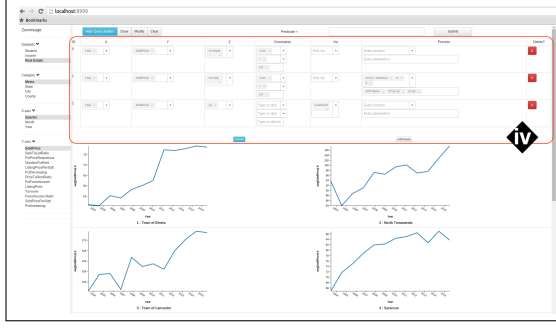


Figure 12: Custom query builder with bar charts

of candidate visualizations by letting us derive a Cartesian product for the z-axis. Table 33 gives an example of using an additional Z column. The query is used to slice the data in two ways: one in terms of the products and one in terms of the year. The resulting set of visualizations could potentially then be used to generate comparison sales over time visualizations between USA and Canada for items whose sales differed the most between USA and Canada.

X	Y	Z
'year'	'sales'	z1.v1 <- ('product'.{'chair', 'desk'}) 'location'.US

Table 32: A ZQL query which returns a set of sales over year visualizations; one for chairs, one for desks, and one for items sold in the US. (Name is not displayed above.)

X	Y	Z	Z2
'year'	'sales'	v1 <- 'product'.*	v2 <- 'location'.{USA, Canada}

Table 33: A ZQL query which returns the set of sales over year visualizations for each product in USA and Canada (Name is not displayed above.)

A.3 Additional Details on the Constraints Column

Beyond the reasons described previously, there are a few additional ways the constraints column differs from the Z column. When ZQL returns the data which the zenvisage front-end uses to render actual visualizations with, the values for the Z columns are returned as part of the output. This is usually necessary as the Z column often iterates over a set of values, so the values for the Z column must be returned for proper identification as to which part of the returned data belongs to which slice. The results of the Constraints column, however is not returned as an output of ZQL.

Secondly, because of the restrictions we have put on the Constraints column, we can allow the Constraints column to be much more complex than the Z columns. Earlier, we saw that we required extra Z columns to be able to deal with more than one attribute in the z-axis, but with the Constraints column, all of those constraints can be combined in one single column. The best way to think about the Constraints column is to imagine that the expression in the Constraints column will simply be added conjunctively to the WHERE clause of the translated SQL queries. In fact, the syntax for the Constraints column has been adjusted so that the expression can be taken from a ZQL table and directly added to SQL. For example, there is no need to quote attributes in the Constraints column. As a result, the set of possible values for the Constraints clause is roughly equal to the set of possible expressions for the WHERE clause in SQL. An example which makes extensive use of the Constraints column is given by Table 34.

Name	X	Y	Constraints
*f1	'time'	'sales'	product='chair' AND zip LIKE '02\d{3}'

Table 34: A ZQL query for a sales over time visualization for chairs sold in the US in zip codes between 02000 and 02999.



Figure 13: Custom query builder with scatterplots

A.4 Additional Details on the Viz Column

Like other columns, the Viz column also supports sets of values and iteration over those sets. Table 35 refers to a ZQL query which iterates over various binning sizes for the x-axis. Table 36 iterates over the types of visualizations using t1. Note, unlike the Z column, we do not need to create a variable to iterate over the summarization even though the visualization type changes, since the same summarization is valid for both types.

Although the bar chart and dot plot does not take in any parameters, other types of charts, such as the box plot, may take in additional parameters (e.g., to determine where the whisker should end); these additional parameters may be added as part of the summarization.

A.5 Additional Details on the Name Column

In addition to using the X/Y/Z columns to define visual components, the user may also use the Name column to derive visual components based on other visual components. For example, in Table 37, visual component f3 is defined with $f3 = f1 + f2$; this creates a visual component by appending the visualizations in f1 with the visualizations in f2 and assigns the new name variable f3 to resulting visual component. Other valid operations include (i) $f3 = f1 - f2$: where f3 refers to the set of visualizations in f1 with the exception of the set of visualizations in f2, (ii) $f2 = [f1[i]]$: where f2 refers to the ith visualization in f1, and (iii) $f2 = f1[i:j]$: where f2 refers to the set of visualizations starting from ith visualization to the jth visualization in f1. These operations are useful if the user wants to throw away some visualizations, or create a new larger set of visualizations from smaller sets of visualizations.

After a visual component has been derived using the Name column, the user may also define axis variables in the X, Y, and Z columns using the special _ symbol to bind to the derived visual component. For example in Table 37, v2 is defined to be the iterator which iterates over the set of product values which appear in derived visual component f3; in this case, v2 iterates over all possible products. y1 is defined to be the iterator over all the values in the Y column of f3. Although in the case of Table 37, the only value y1 takes on is 'sales', y1 and v2 are considered to be declared together, so the iterations for y1, v2 will look like: [('sales', 'chair'), ('sales', 'table'), ...]. Also in this case, the variable y1 is not used, however, there may be other cases where it may be useful to iterate over multiple axis variables. The defined axis variables can then be used to create other visual components or within the Process column as shown in the 4th row of Table 37.

There is a subtle point here: the semantics of the name and axis variables for derived visual components are the opposite of what they were for the regular visual components. For regular visual components, which did not use an expression for the Name column, the name variables were bound to the resulting visual components created by combining the different axis variables. However, for de-

Name	X	Y	Viz
*f1	'weight'	'sales'	s1 <- bar.{(x=bin(20), y=agg('sum')), (x=bin(30), y=agg('sum')), (x=bin(40), y=agg('sum'))}

Table 35: A ZQL query which returns the bar charts of overall sales for different weight classes, for varying weight class sizes.

Name	X	Y	Viz
*f1	'weight'	'sales'	t1 <- {bar, dotplot}.{(x=bin(20), y=agg('sum'))}

Table 36: A ZQL query which returns the set of bar chart and dot plot of overall sales for different weight classes.

rived visual components, the visual components have already been created, and the axis variables merely provide a way to iterate over the derived visual component, and refer to them in subsequent rows of the ZQL table.

A.6 Additional Details on the Process Column

Even with the primitives defined in the paper so far, sometimes the user would like to be able to write her own. ZQL supports user-defined functions that are executed by the *zenvisage* back-end. User-defined functions may take in name and axis variables and perform whatever computation is necessary; *zenvisage* treats them as black boxes. However, users are encouraged to use the primitives defined by ZQL as they allow the *zenvisage* back-end more opportunities for optimization.

Furthermore, although visual components typically outnumber processes, there may occur cases in which the user would like to specify multiple processes in one line. To accomplish this, the user simply delimits each process with a comma and surrounds each declaration of variables with parentheses. Table 38 gives an example of this.

B. QUERY EXECUTION: NAIVE TRANSLATION LISTING

Here, we list the pseudocode for the naive translation for query execution for reference.

Listing 1: Pseudocode for the compiled query in Table 14

```
# Row 1
v1_range = P,
f1 = make_array(1, size(v1_range))
for v1 in [0 .. size(v1_range)]:
    f1[v1] = sql("SELECT year, SUM(sales)
                WHERE product='%s' and location='US'
                GROUP BY year ORDER BY year'",
                v1_range[v1])
v2_range = []
for v1 in [0 .. size(v1_range)]:
    if T(f1[v1]) > 0:
        v2_range.append(v1_range[v1])

# Row 2
f2 = make_array(1, size(v1_range))
for v1 in [0 .. size(v1_range)]:
    f2[v1] = sql("SELECT year, SUM(sales)
                WHERE product='%s' and location='UK'
                GROUP BY year ORDER BY year'",
                v1_range[v1])
v3_range = []
for v1 in [0 .. size(v1_range)]:
    if T(f2[v1]) < 0:
        v3_range.append(v1_range[v1])

# Row 3
v4_range = union(v2_range, v3_range)
f3 = make_array(1, size(v4_range))
for v4 in [0 .. size(v4_range)]:
    f3[v4] = sql("SELECT year, SUM(profit)
```

```
WHERE product='%s'
GROUP BY year ORDER BY year")),
v4_range[v4])
```

return f3

C. ADDITIONAL COMPLETE EXAMPLES

To demonstrate the full expressive power of ZQL, we present four realistic, complex example queries. We show that even with complicated scenarios, the user is able to capture the insights she wants with a few meaningful lines of ZQL.

Query 1. The stapler has been one of the most profitable products in the last years for GlobalMart. The Vice President is interested in learning about other products which have had similar profit trends. She wishes to see some representative sales over the years visualizations for these products.

Table 39 shows what the query that the Vice President would write for this scenario. She first filters down to the top 100 products which have the most similar to profit over year visualizations to that of the stapler’s using the *argmin* in the second row. Then, from the resulting set of products, *v2*, she picks the 10 most representative set of sales over visualizations using *R*, and displays those visualizations in the next line with *f4*. Although the Vice President does not specify the exact distance metric for *D* or specify the exact algorithm for *R*, she knows *zenvisage* will select the most reasonable default based on the data.

Query 2. The Vice President, to her surprise, sees that there a few products whose sales has gone up over the last year, yet their profit has declined. She also notices some product’s sales have gone down, yet their profit has increased. To investigate, the Vice President would like to know about the top 10 products who have the most discrepancy in their sales and profit trends, and she would like to visualize those trends.

This scenario can be addressed with the query in Table 40. The Vice President names the set of visualizations for profit over month *f1* and the sales over month visualizations *f2*. She then compares the visualizations in the two set using the *argmax* and retrieves the top 10 products whose visualizations are the most different. For these visualizations, she plots both the sales and profit over months; *y1 <- {'sales', 'profit'}* is a shortcut to avoid having to separates rows for sales and profit. Note that the Vice President was careful to constrain ZQL to only look at the data from 2015.

Query 3. The Vice President would like to know more about the differences between a product whose sales numbers do not change over the year and a product that has the largest growth in the number of sales. To address this question, she writes the query in Table 41. The first *R* function call returns the one product whose sales over year visualization is most representative for all products; in other words, *v2* is set to the product that has the most average number of sales. The task in the second row selects the product *v3* which has the greatest upward trending slope *T* for sales. Finally, the Vice President tries to finds the y-axes which distinguish the two products the most, and visualizes them. Although we know *v2* and *v3* only contain one value, they are still sets, so *argmax* must iterate over them and output corresponding values *v4* and *v5*.

Name	X	Y	Z	Process
f1	'year'	'sales'	v1 <- 'product'.(* - 'stapler')	v3 <- argmax _{v2} k = 10]D(f3, f4)
f2	'year'	'sales'	'stapler'	
f3=f1+f2			y1 <- 'product'._	
f4	'year'	'profit'	v2	
*f5	'year'	'sales'	v3	

Table 37: A ZQL query which returns the sales over years visualizations for the top 10 products which have the most different sales over years visualizations and profit over years visualizations.

Name	X	Y	Z	Process
-f1				(v2 <- argmax _{v1} [k = 1]D(f1, f2)), (v3 <- argmin _{v1} [k = 1]D(f1, f2))
f2	'year'	'sales'	v1 <- 'product'.*	
*f3	'year'	'sales'	v2	
*f4	'year'	'sales'	v3	

Table 38: A ZQL query which returns the sales over years visualizations for the product that looks most similar to the user-drawn input and most dissimilar to the user-drawn input.

Name	X	Y	Z	Viz	Process
f1	'year'	'profit'	'product'. 'stapler'	bar.(y=agg('sum'))	v2 <- argmin _{v1} [k = 100]D(f1, f2) v3 <- R(10, v2, f3)
f2	'year'	'profit'	v1 <- 'product'.(* \ { 'stapler' })	bar.(y=agg('sum'))	
f3	'year'	'sales'	v2	bar.(y=agg('sum'))	
*f4	'year'	'sales'	v3	bar.(y=agg('sum'))	

Table 39: The ZQL query which returns 10 most representative sales over year visualizations for products which have similar profit over year visualizations to that of the stapler's.

Name	X	Y	Z	Constraints	Viz	Process
f1	'month'	'profit'	v1 <- 'product'.*	year=2015	bar.(y=agg('sum'))	v2 <- argmax _{v1} [k = 10]D(f1, f2)
f2	'month'	'sales'	v1	year=2015	bar.(y=agg('sum'))	
*f3	'month'	y1 <- {'sales', 'profit'}	v2	year=2015	bar.(y=agg('sum'))	

Table 40: The ZQL query which returns the sales over month and profit over month visualizations for 2015 for the top 10 products which have the biggest discrepancies in their sales and profit trends.

Name	X	Y	Z	Viz	Process
f1	'year'	'sales'	v1 <- 'product'.*	bar.(y=agg('sum'))	v2 <- R(1, v1, f1)
f2	'year'	y1 <- M	v2	bar.(y=agg('sum'))	v3 <- argmax _{v1} [k = 1]T(f1)
f3	'year'	y1	v3	bar.(y=agg('sum'))	y2, v4, v5 <- argmax _{y1, v2, v3} [k = 10]D(f2, f3)
*f4	'year'	y2	v6 <- (v4.range v5.range)	bar.(y=agg('sum'))	

Table 41: The ZQL query which returns varying y-axes visualizations where the following two products differ the most: one whose sales numbers do not change over the year and another which has the largest growth in the number of sales.

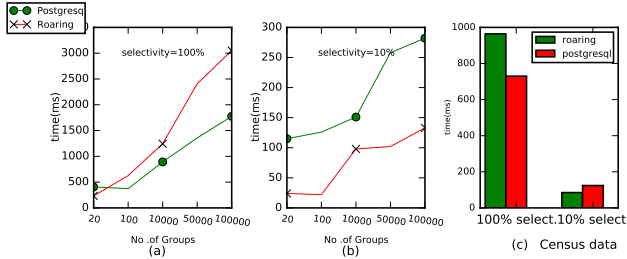


Figure 14: Comparing RoaringDB vs PostgreSQL execution time a,b) on varying selectivities and no. of groups on synthetic dataset c) on varying selectivities on a real dataset

Query 4: Finally, the Vice President wants to see a pair of dimensions whose correlation pattern (depicted as a scatterplot) is the most unusual, compared to correlation patterns of other pairs of attributes. To address this question, she writes the query in Table 42. She keeps the Z column empty as she does not want to slice the data. Both X and Y refer to a set M consisting of all the attributes in the dataset she wishes to explore. The task in the second row selects the X and Y attributes whose sum of distances from other visualizations (generated by considering all pairs of attributes) is the maximum.

D. EVALUATING BACK-END DATABASES

In this experiment, we compare the performance between our two database back-ends: PostgreSQL and the in-memory Roaring Bitmap Database. To make the comparison fair, we indexed both databases similarly: we indexed all categorical attributes in the bitmap database, while for PostgreSQL, we applied multi-column indexing on all categorical attributes.

We use a simple aggregate query of the following form to measure the performance of two database systems under two levels of

query selectivity: 10% and 100%. This query is representative of a vast majority of SQL queries generated while processing ZQL queries:

```
SELECT X, F(Y), Z FROM table
WHERE P1=p1 AND P2=p2
GROUP BY Z, X ORDER BY by Z, X
```

For each trial, we chose random categorical attributes for the values of X, Z, P1, P2, and a random measure attribute for Y. We also randomly determined the values for p1 and p2. In addition, we ran a version of the SQL query above without any predicates for the 100% selectivity trials.

We compare the performance of two databases with varying the number of groups and selectivity of the query:

10% selectivity. When the selectivity of the query is low, Roaring Bitmap's fast bitwise operations helps in identifying the rows used in the aggregation. We see in Figure 14 (b) that this allows our Roaring Bitmap Database to perform 30-80% better than PostgreSQL, regardless of groups.

100% selectivity. However, when the selectivity is 100%, the bitmap indexes buy us nothing as we need to look at every row in the database. Figure 14 (a) shows us that Roaring Bitmap Database still outperforms than PostgreSQL on a small number of groups. However, as the number of groups increases, the overhead from the hash-lookups required to select the groups becomes too great, and Roaring Bitmap Database ends up performing 30-50% worse than PostgreSQL.

We observe similar results on the real dataset as shown in Figure 14 (c).

E. USER STUDY: ADDITIONAL DETAILS ON FINDING 5

Name	X	Y	Z	Viz	Process
f1	x1 <- M	y1 <- M			x3,y3 <- $\text{argmax}_{x_1,y_1}[k=1]\text{sum}_{x_2,y_2}D(f_1,f_2)$
f2	x2 <- M	y2 <- M			
*f3	x3	y3		scatterplot	

Table 42: The ZQL query which returns scatter plot visualization between a pair of attributes whose pattern is most unusual, i.e very different from the patterns made by any other pair of attributes in M .

Participant P6's python code implementation for the task in Table 12:

```
[language=python]
import pandas
import numpy as np
def generate_maps(date_list, d, Y, Z):
    d = d[d['State']==Z][np.append(date_list, Y)]
    maps = {}
    for id, item in d.iterrows():
        date = ""
        for k in date_list:
            date += str(item[k])
        if date not in maps:
            maps[date] = []
            maps[date].append(item[Y])
    maps = dict([(k, np.mean(v)) for k, v in maps.items()])
    return maps
def filter(d, X, Y, Z):
    """
    X : Month, Year, Quater
    Y : SoldPrice, ListingPrice, Turnover_rate
    Z : State Name such as CA
    """
    maps = {}
    if X == 'Year':
        date_list = ['Year']
    elif X == 'Quater':
        date_list = ['Year', "Quater"]
    elif X == 'Month':
```

```
        date_list = ['Year', "Quater", "Month"]
    return generate_maps(date_list, d, Y, Z)
def mapping(map1, map2):
    """ calculate distance"""
    t = 0.0
    for k, v in map1.items():
        t += (map2[k] - v) * (map2[k] - v)
    return t

if __name__ == "__main__":
    import matplotlib.pyplot as plt
    import numpy.linalg as LA
    d = pandas.read_csv("./tarique_data")
    XSet = ["Year", "Quater", "Month"]
    YSet = ["SoldPrice", "ListingPrice", "Turnover_rate"]
    result = [(X, Y, mapping(filter(d, X, Y, 'CA'),
    filter(d, X, Y, 'NY')))) for X in XSet for Y in YSet]
    best_x, best_y, difference = sorted(result,
    cmp=lambda x, y: -cmp(x[2],y[2]))[0]
    CA, NY = filter(d, best_x, best_y, 'CA'),
    filter(d, best_x, best_y, "NY")
    xset = CA.keys()
    xset.sort()
    y_CA, y_NY = [CA[x] for x in xset],
    [NY[x] for x in xset]
    plt.plot(range(len(xset)), y_CA, label='CA')
    plt.plot(range(len(xset)), y_NY, label='NY')
    plt.legend()
    plt.show()
```